PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F

A2

(11) International Publication Number: WO 00/28397

(43) International Publication Date: 18 May 2000 (18.05.00)

(21) International Application Number: PCT/US99/26474

(22) International Filing Date: 10 November 1999 (10.11.99)

(30) Priority Data:
09/189,763 10 November 1998 (10.11.98) US

(71)(72) Applicant and Inventor: FARROW, Robert, C. [US/US]; 9754 Vinewood Drive, Dallas, TX 75228–3772 (US).

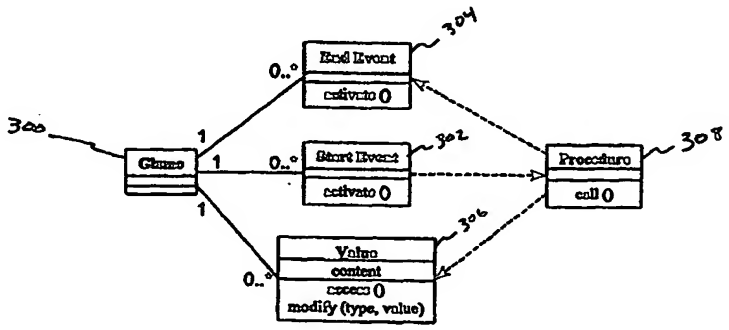
(72) Inventor: STANFORD, Paul, H.; Suite 106, 10530 Stone Canyon, Dallas, TX 75230 (US).

(74) Agent: SCOTT, Russell, C.; Akin, Gump, Strauss, Hauer & Feld, LLP, Suite 1900, 816 Congress Avenue, Austin, TX 78701 (US).

(81) Designated States: AU, CA, CN, IN, JP, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: AN INTRINSICALLY PARALLEL PROGRAMMING SYSTEM AND METHOD USING PROCEDURAL OBJECTS



(57) Abstract

A system and method of programming using procedural objects referred to as gizmos, where each gizmo includes an underlying procedure and at least one associated component. Each component is a value or an event. The gizmo instances are preferably self-contained copies of underlying stored gizmos. A user creates new gizmos with new functionality by linking or "stitching" components of gizmo instances together, where stitched components become equivalent. Value components are stitched to other value components and event components are stitched to other event components. When any stitched value is updated by the user or by a procedure, the other values stitched therewith are updated automatically. Also, when any stitched event is activated, the other events stitched therewith are automatically activated. The combined stitching is referred to as a "fabric", which is a collection of related stitches. In a Graphic User Interface environment, the user retrieves instances of stored gizmos from memory that are displayed in an editing environment referred to as a play space. The play space contains an entirely independent gizmo function including copies of any primitive gizmos and underlying gizmos. Each gizmo includes a gizmo face representing the underlying gizmo and graphic representations of each component. A gizmo face is an abstraction interface that is created with new components that are stitched to components of the existing gizmos. The new gizmo may then be stored, and instances of the new gizmo may be retrieved and executed as desired or used to create yet other new gizmos.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

TITLE:
AN INTRINSICALLY PARALLEL PROGRAMMING SYSTEM AND METHOD USING PROCEDURAL OBJECTS

Robert C. FARROW, Paul H. STANFORD

FIELD OF THE INVENTION

The present invention relates to an intrinsically parallel programming model for a computer system, and more particularly to an intrinsically parallel programming system and method using procedural objects.

DESCRIPTION OF THE RELATED ART

Computers have become an integral and important part of our daily lives. For many tasks, computers are faster, more accurate and more efficient than humans. Many common tasks performed today by computers are virtually impossible for humans to perform. Computers often enable humans to perform tasks faster and more efficiently.

Nonetheless, the capability of any computer is strictly dependent on humans. An inevitable and inherent part of a computer is programming, since a computer must be properly programmed to perform any task. The concept of programming is relatively simple, and many of the most complex computer programs comprise simple steps performed at a high rate of speed. Nonetheless, the actual practice of programming has traditionally not been a simple task. Computers are precise, exacting and unforgiving machines that often fail in the event of the slightest programming error. Programming usually required mastery of at least one high level programming language, such as Basic, Cobol, Pascal, Fortran, C, C++, OLE, etc. Each programming language was designed for one or more particular programming tasks, and typically had one or more strengths and one or more weaknesses. Few programming languages, if any, were straightforward, and each generally required a significant amount of time and study to learn. Every traditional programming language, without exception, has been subject to

relatively strict rules that had to be followed exactly. Even simple programming tasks have been difficult. Thus, an experienced programmer was usually required to write relatively simple programs, which substantially increases time and expense associated with programming computers.

Nonetheless, programming was rarely simple even for experienced programmers. The typical programmer had to be very careful and had to take meticulous steps to set up the proper parameters, constants, variables, types, etc., and had to follow the appropriate syntax to achieve a working program. This was often achieved by trial and error. Even when it was believed that the program functioned properly, software bugs were common. This was especially true for the most popular, off-the-shelf computer programs typically sold in retail stores.

Even if one or more programming languages were mastered, the resulting code usually had several limitations. Few programming languages, if any, result in intrinsically parallel code. Most programming languages and the resulting code are hierarchical in nature and sequential in operation. The structure is forced by the top-level down approach to many high-level languages. The main code typically includes calls to functions or subroutines that require parameter passing. Most languages and software code, including object-oriented languages, are oriented around data-structures. Also, most languages are text-based and dependent upon variable, parameters and names and often requires name compatibility. Although many programming languages allow certain levels of modularity, few programming languages, if any, provide a mechanism for truly modular and reusable functionality.

It is desired to provide a system and method for programming a computer that overcomes the above-listed limitations of traditional programming languages. It is desired to provide an intuitive and more powerful system and method for programming a computer that requires little if any programming experience or expertise. It is desired to reduce or eliminate name dependence and compatibility requirements and to provide a mechanism for truly modular and reusable functionality.

SUMMARY OF THE INVENTION

A computer programming system according to the present invention includes an executor, a plurality of components, where each component is associated with input/output

and comprising either a value or an event, a plurality of procedures, where each procedure is associated with at least one component and includes at least one input event, and at least one combiner, where each combiner links any values together to form stitched values and links any events together to form stitched events. The executor detects activation of a first input event, executes any of the procedures upon activation of any associated input events, updates stitched values and activates stitched events. Each of the plurality of procedures, if and when executed by the executor, may receive any associated input values, detect any associated input events, and, as determined by the associated procedure, update any associated output values and activate any associated output events.

The computer programming system may further include a plurality of procedural objects, where each procedural object includes at least one procedure and at least one component associated with that procedure. The at least one combiner may link at least one component of a first procedural object to at least one component of a second procedural object. Each combiner may further link at least one output event of a first procedural object to at least one input event of a second procedural object and to at least one input event of a third procedural object, where the second procedural object includes a first procedure and the third procedural object includes a second procedure. In this manner, during execution, the first procedure and second procedure may effectively be executed simultaneously by the executor.

At least one of the procedural objects may include at least one input component and at least one output component. The output component may comprise an output value that depends upon the input component when the procedure of the procedural objects is executed. Alternatively, the output component may comprise an output event that is activated depending upon the input component when the procedure is executed. Each of the plurality of procedural objects may include at least one component that is linked with at least one component of any other of the plurality of procedural objects. Each of the plurality of procedural objects may be self-timed and coherent. Further, each of the plurality of procedural objects, when executed, may operate independently with respect to every other of the plurality of procedural objects.

The executor may operate as an interpreter in an interpretive mode. Alternatively, the executor may include a compiler that generates executable program code based on the plurality of procedures, the plurality of components and linking of components via the combiner(s).

A computer system according to the present invention includes a processor, at least one input device and a memory that stores data and program code for execution by the processor. The program code includes a plurality of component objects, where each component object is either one of a value or an event, a plurality of procedure modules, where each procedure module is associated with at least one component object, and an edit utility that when executed by the processor, enables a user to retrieve any of the plurality of procedure modules and associated component objects, to stitch component objects of retrieved procedure modules together to create a new procedural object and to store the new procedural object in the memory. The processor, when executing the new procedural object, executes any of the plurality of procedure modules included in the new procedural object, updates any stitched values if any one value of the any stitched values is updated and activates any stitched events if any one event of the any stitched events is activated.

In a graphic user interface (GUI) environment, the computer system may further include a display, where the edit utility displays graphic representations of selected procedure modules and component objects on the display. Also, the edit utility displays graphic representations of manipulations of the input device(s) by the user on the display to enable the user to interactively stitch component objects of the selected procedure modules together to create the new procedural object. Further, the edit utility may enable a user to associate a procedure graphic with the new procedural object, where the executor displays the procedure graphic and enables the user to execute the new procedural object by interacting with the procedure graphic via the input device(s).

The memory may store a plurality of predetermined procedural objects, where each procedural object includes at least one procedure module and at least one component object associated with the procedure module(s). For example, the plurality of procedural objects may comprise a library of predefined procedural object primitives, where each primitive performs at least one basic programming function. The edit utility may enable a user to

associate a procedure block with the new procedural object, where the executor enables the user to execute the new procedural object by accessing the procedure block via the input device(s). For example, the edit utility may enable the user, via the input device(s), to retrieve instances of any of the predetermined procedural objects and an instance of the new procedural object as represented by the procedure block and to stitch component objects of the new procedural object with component objects of other retrieved procedural objects to create a second new procedural object and to store the second new procedural object in the memory.

For a GUI environment including a display, the edit utility displays graphic representations of each retrieved procedural object including graphic representations of the associated component objects. Also, the edit utility displays representations of inputs and manipulations of the input device(s) by the user on the display to enable the user to interactively stitch component objects of the each retrieved procedural objects together to create the new procedural object. The edit utility may further display a procedure graphic and enable a user to associate the procedure graphic with the new procedural object. Also, the executor displays the procedure graphic and enables the user to execute the new procedural object by interacting with procedure graphic via the input device(s). Further, the edit utility enables the user to retrieve and display the procedure graphic and to stitch component objects of the new procedural object represented by the procedure graphic with component objects of other retrieved procedure module instances to create a second new procedural object and to store the second new procedural object in the memory.

The computer system may further include an interpreter that interprets the plurality of procedure modules and stitched components of the new procedural object when executed. The processor executes the new procedural object via the interpreter. Alternatively, a compiler is included that compiles the new procedural object into executable program code, where the processor accesses and executes the program code.

In a more specific embodiment, each procedural object is referred to as a "gizmo", where each gizmo includes an underlying procedure or procedure module and at least one associated component. In a GUI environment, the user retrieves instances of stored gizmos from memory that are displayed on a monitor or the like. The retrieved gizmo instances are

displayed in an editing environment referred to as a "play space". The gizmo instances are preferably self-contained copies of the underlying stored gizmos. Each gizmo includes a gizmo face representing the underlying gizmo and graphic representations of each component. For example, value components may be represented by a value graphic widget or graphic box that enables the user to enter a value into the box, such as an alphanumeric value or the like. Event components may be represented by graphic buttons, which may be activated by the user via one or more input devices (such as a mouse, keyboard, etc.), by the underlying procedure or by other stitched events.

For example, an add gizmo may include two input value boxes, an output value box, a start event button and an end event button. The user enters first and second numbers to be added into the two input value boxes, respectively, and presses the start event button. The underlying procedure adds the two numbers to obtain a sum value, places a sum value into the output value box, and activates the end event button. The end event button is graphically displayed as being activated using any type of graphic technique, such as highlighting, color changes, brightness changes, added or modified symbols, etc. The add gizmo may further include an error end event button that is activated by the underlying procedure in the event of an error, such as detection of an invalid input value.

The user creates new gizmos or new gizmo functions by linking or "stitching" components of instances of existing gizmos together, where stitched components become equivalent. Value components are stitched to other value components and event components are stitched to other event components. When any stitched value is updated by the user or by a procedure, the other values stitched therewith are updated automatically. Also, when any stitched event is activated, the other events stitched therewith are automatically activated. The combined stitching is referred to as a "fabric". A fabric is one way to conceptualize a combiner object, which is a collection of related stitches. In an embodiment described herein, the fabric generally comprises combiner instances that combine one or more values or one or more events. In this manner, the user creates a new gizmo with new functionality by stitching together components of existing gizmos. A new gizmo face is created with new components that are stitched to components of the existing gizmos. The new gizmo may then be stored, and instances of the new gizmo may be retrieved and executed as desired or used to create yet other new gizmos. For example, an instance of a newly created gizmo may be retrieved into a

new play space and executed or stitched with other gizmo instances to create another new gizmo. When retrieved, the gizmo face of the new gizmo is displayed and represents the underlying gizmo. In the preferred embodiment, the instance is a self-contained copy of the stored gizmo and thus is independent of its stored parent gimzo.

A library of predetermined or predefined gizmos may be provided, where each performs a particular programming function. Such programming functions may be very basic but may also be very complex. Such programming functions may include typical programming tasks, such as number value copying, timing delay, sequential operation functions, random number generation, audio visual effects, file access, gateways to other programs, etc. Such programming functions may also include mathematical functions, such as addition, subtraction, multiplication, division, negation, etc. Such programming functions may further include any type of logic functionality. It is noted, however, that traditional boolean logic functions, such as logic AND, logic OR, logic negation, logic addition or subtraction, etc., are replaced with event logic functions, such as event AND, event OR, event AFTER, etc. For example, a gizmo may be defined with a plurality of input events and an output event that is activated after all of the input events are activated to achieve AND functionality. Likewise, OR functionality may be achieved with a similar gizmo by activating the output event after any one of the plurality of input events are activated. In general, looping or branching constructs of traditional programming systems are replaced by the construction of a set of gizmos stitched in a loop or in a branch.

The user may use predefined gizmos as building blocks to create a new gizmo with new functionality as described above. The new gizmo may be added to an existing library of gizmos or to a new library. Each predefined and/or newly defined gizmo is thus available as a building block for new functionality. In this manner, an intrinsically programming system according to the present invention is virtually unlimited so that any programming task may be achieved by almost anyone regardless of programming experience.

Of note, while editing or programming gizmo functionality in a play space, the user may test the new gizmo program by activating a start event to "execute" the new gizmo. Since each gizmo instance is a self-contained copy of the underlying parent gizmo, the play space contains an entirely independent gizmo function including copies of any primitive

gizmos and underlying gizmos. In this manner, the gizmo is executed in an interpretive mode to enable the user to test the functionality. A new gizmo may be compiled by a compiler to optimize the gizmo by reducing or eliminating redundant code and/or optimizing operation of various parts of the code. The compiled gizmo thus becomes a self-contained gizmo program that performs the same function in a more efficient (and usually faster) manner. Values of components when a gizmo is saved become initial values of the instances of that gizmo.

A system and method according the present invention enables intrinsically parallel programs to be easily written by any user. The programs are easily executed by a distributed computation system. For example, the distributed computation system includes a network and a plurality of computers participating in the network. Each computer comprises a corresponding one of a plurality of processors and a corresponding one of a plurality of memory systems. The plurality of computers includes a first computer, which further includes a processor, at least one input device and a memory system that stores program code for execution by the processor. The processor executes the new procedural object by distributing the plurality of procedure modules included in the new procedural object among the distributed processing system. In this manner, each processor of the distributed processing system executes a distributed portion of the new procedural object.

A method of programming a computer system according to the present invention includes retrieving instances of selected procedural objects from a plurality of procedural objects, where each of the plurality of procedural objects includes an associated procedure and at least one associated input/output component. The method further comprises stitching together components of the instances of the selected procedural objects including at least one start event that invokes an associated procedure of the selected procedural objects when the start event is activated. The method further comprises storing a new procedural object that includes the instances of the selected procedural objects and fabric representing components that are stitched together. The stitching may comprise including a combiner instance for each stitch that links components of the instances of the selected procedural objects . The retrieving may include displaying a graphic of each of the instances of selected procedural objects, and, for each of the instances of selected procedural objects, including a graphic for each associated component. The stitching may include displaying stitch graphics to facilitate a visual representation of stitching components and of stitched components.

The method may further comprise a processing means executing the new procedural object by interpreting the new procedural object to detect a start event, to execute any associated procedure of the instances of selected procedural objects initiated by the associated input event, to update stitched values and to activate stitched events. Alternatively, the method may comprise compiling the new procedural object into an executable program in a similar manner as described previously. The method may further comprise retrieving the new procedural object, and, in an edit mode, modifying the fabric by modifying the stitches between any components.

Gizmo procedural objects as described herein are self-timed coherent components, each with clearly defined, non-interfering functionality. A gizmo can be a stand-alone program, but also may be combined with other gizmos to extend their joint functionality. Further, combined gizmos can be encapsulated behind another gizmo object to define a new gizmo. Gizmo procedural objects as described herein are interesting and useful as a method for composing component-based programs. Gizmo programs are designed, from the user-standpoint, in a hierarchical manner in which there are an unlimited number of levels of such hierarchy. The present invention provides, however, automatic loading of intermediate levels of hierarchy as needed. Further, the present invention provides the ability to encapsulate the entire hierarchy below a given user-defined abstraction, for ease of management, security, stability of operation and improved efficiency. A compiler may be used to accelerate operation of encapsulated abstraction by optimizing the storage and execution of the gizmo. A GUI embodiment enables keyboard entry of input values, mouse-activated buttons, and responses with visual display of output values, change of graphical display, sound effects, file modification, etc.

A method and system according to the present invention enables natural production and distribution of domain specific libraries of basic and composite gizmos, designed and optimized for use in such domains. Such domains may include, but are not limited to, education, real estate appraisal, accounting, contract analysis, astronomy, career counseling, home diagnosis, contingency planning, entertainment, truck scheduling, inventory control, cash flow management, contact structure, market trend analysis, address maintenance, industrial control, etc.

It is now appreciated that a method and system according to the present invention provides many programming benefits, including integration, concurrency, quality and modularity. Gizmo procedural objects are designed to be program building blocks. There are no restrictions on how simple or how complex a gizmo's behavior may be. Gizmo programming as described herein does not intrinsically limit the programs that are constructed, where the only limitations are the available primitives and the imagination of the user. Gizmos are building blocks that are used to piece together a program which promotes good programming practice. Primitive gizmos may further be implemented to interface with other software. This allows the user to combine the capabilities of many software packages into a custom program. Conventional system interface protocols (like DDE, IPC, OLE, etc.) may be implemented in gizmos to create foreign interfaces.

Each gizmo behaves as an autonomous entity, in communication with other gizmos with which it is combined. The memory for each gizmo is treated as separate, but is in communication with the memory of other gizmos. This symmetry makes it easy to apply the gizmo model to a network application environment. Communicating gizmos could be memory resident on separate machines connected by a network. The user of the gizmo program need not be concerned with how or where the other gizmos in a system are implemented.

While the user need only be concerned with the apparent conceptual model of gizmo programming, the opportunity for optimization still exists. When an optimizing compiler is added to a gizmo development implementation, the compiler has all the important aspects of strongly typed conventional languages and much more information about programmer intent. Compilation takes place on a gizmo abstraction hierarchy. The top gizmo class is the one which represents the root of the tree. The composed gizmos which implement that class are combined and optimized for speed and space efficiency. This also produces a stable encapsulation of the entire hierarchy and thus aids change-management protocols.

The features of gizmo programming naturally promote quality results. Unlike conventional text-based languages there is no description redundancy or syntactic errors. Gizmo programs are constructed in a way to reduce software failures. Further, the

characteristics of visual programming allows for very effective testing and debugging techniques.

Gizmo programs achieve modularity by being divided up into well defined blocks with clear and consistent interfaces. These interfaces help isolate details of the implementation and reveal important program component relationships. Initially the benefit of modularity makes it easier to modify and maintain a program. Ultimately, modules may be reused by other programmers and programs. Reuse provides for improved programmer productivity and product quality. Gizmo programming according to the present invention encourages good modularity. Each gizmo represents a module that can be combined with other modules to define a program. The added benefit is that a gizmo can be initiated without any context. Each gizmo is instantiated in a way that makes it function as a stand alone program. Even if incorrect values are provided, the gizmo responds with the activation of an error end event.

Another important gizmo reuse feature is the lack of dependence on names. In windowing or GUI forms of gizmo implementation, naming is merely a decoration. The user refers to the elements of a program by pointing and relates them by stitching. Name compatibility is no longer a reuse consideration. The gizmo paradigm provides a mechanism for truly modular reusable functionality. It reveals the true utility of the computer directly to users, and results in unfettered creativity and cooperation among users.

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the present invention can be obtained when the following detailed description of the preferred embodiment is considered in conjunction with the following drawings, in which:

Figure 1 is a block diagram of an exemplary computer system that is used to illustrate various aspects of a programming system implemented according to the present invention.

Figure 2 is a block diagram of a network system that communicatively couples a plurality of computer systems or computing devices together via a communication medium.

Figure 3 is a block diagram according to the Unified Modeling Language (UML) of a procedural object or "gizmo" including associated and dependency relationships illustrated.

Figure 4 is a block diagram in UML format of an exemplary activation sequence of a procedural object or gizmo that is operating according to the principles of the present invention.

Figure 5 is a block diagram in UML format of an exemplary combiner that illustrates relationships between the combiner and event components and value components.

Figure 6 is a block diagram in UML format of an exemplary activation sequence when the combiner of Figure 5 is used to combine multiple event components and associate a procedure module.

Figure 7 is a block diagram in UML format of an exemplary modification sequence when the combiner of Figure 5 is used to combine multiple value components.

Figure 8 is a block diagram in UML format of an exemplary gizmo class illustrating gizmo relationships between zero or more value components and zero or more event components.

Figure 9 is a block diagram in UML format of an exemplary gizmo container that is used to store gizmo instances that are accessible for use in constructing a gizmo class.

Figure 10 is a block diagram in UML format illustrating steps of a gizmo instantiation sequence.

Figures 11A and 11B are illustrations of an exemplary graphical representation of a procedural object or a gizmo instance or simply a gizmo as displayed in a graphical user interface (GUI) environment.

Figures 12A, 12B and 12C are illustrations of the exemplary GUI representation of the gizmo instance of Figure 11A during a stitching operation.

Figure 13 is an illustration of an exemplary graphic play space that illustrates gizmo composition by using existing gizmos to create a new gizmo with new functionality.

Figure 14 is an illustration of another graphic play space according to an embodiment of the present invention to illustrate stitches between gizmo instances to create or "program" new functionality.

Figures 15 and 16 are graphical illustrations of editing modes available in a programming system according to the present invention.

Figure 17 shows the graphic play space of Figure 14 to illustrate the gizmo instances with a new stitch that was created during the editing of the gizmo instances of the play space.

Figure 18 is an illustration of a graphic play space according to an embodiment of the present invention to illustrate exemplary stitches between gizmo instances to create or "program" iterative functionality in the play space.

Figures 19A - 19D are illustrations of an exemplary graphic play space that illustrate an example of iterative/conditional gizmo composition by using existing gizmos to create a new gizmo with new functionality.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure 1 is a block diagram an exemplary computer system 100 that is used to illustrate various aspects of a programming system implemented according to the present invention. The computer system 100, although other systems are clearly possible, is preferably an IBM-compatible, personal computer (PC) system or the like, and includes a motherboard and bus system 102 coupled to at least one central processing unit (CPU) 104 and a memory system 106. The motherboard and bus system 102 includes any kind of bus system configuration, such as any combination of a host bus, one or more peripheral component interconnect (PCI) buses, an industry standard architecture (ISA) bus, an extended ISA (EISA) bus, microchannel architecture (MCA) bus, etc., along with corresponding bus driver circuitry and bridge interfaces, etc., as known to those skilled in the art. The CPU 104 preferably incorporates any one of several microprocessors, such as the 80486, Pentium™, Pentium II™, etc. microprocessors from Intel Corp., or other similar type microprocessors such as the K6 microprocessor by Advanced Micro Devices, and supporting external circuitry typically used in PCs. The external circuitry preferably includes an external or level two (L2)

cache or the like (not shown). The memory system 106 may include a memory controller or the like and be implemented with one or more memory boards (not shown) plugged into compatible memory slots on the motherboard, although any memory configuration is contemplated.

The computer system 100 includes one or more output devices, such as speakers 109 coupled to the motherboard and bus system 102 via an appropriate sound card 108 and a monitor or display 112 coupled to the motherboard and bus system 102 via an appropriate video card 110. One or more input devices may also be provided such as a mouse 114 and a keyboard 116, each coupled to the motherboard and bus system 102 via appropriate controllers (not shown) as known to those skilled in the art. A storage system 120 is coupled to the motherboard and bus system 102 and may include any one or more data storage devices, such as one or more disk drives including floppy and hard disk drives, one or more CD-ROMs, one or more tape drives, etc. Other input and output devices may also be included, as well as other types of input devices including a microphone, joystick, pointing device, voice recognition, etc. The input and output devices enable a user to interact with the computer system 100 for purposes of programming, as further described below.

The motherboard and bus system 102 may be implemented with at least one expansion slot 122, which is configured to receive compatible adapter or controller cards configured for the particular slot and bus type. Typical devices configured as adapter cards include network interface cards (NICs), disk controllers such as a SCSI (Small Computer System Interface) disk controller, video controllers, sound cards, etc. The computer system 100 may include one or more of several different types of buses and slots, such as PCI, ISA, EISA, MCA, etc. In the embodiment shown, a network interface controller adapter card 124 is shown coupled to the slot 122 for interfacing the computer system to a network, if desired.

Other components, devices and circuitry are normally included in the computer system 100 but are not particularly relevant to the present invention and are not shown, although these may be accessed by certain gizmos such as timers. Such other components, devices and circuitry are coupled to the motherboard and bus system 102, such as, for example, an integrated system peripheral (ISP), an interrupt controller such as an advanced programmable interrupt controller (APIC) or the like, bus arbiter(s), one or more system

ROMs (read only memory) comprising one or more ROM modules, a keyboard controller, a real time clock (RTC) and timers, communication ports, non-volatile static random access memory (NVSRAM), a direct memory access (DMA) system, diagnostics ports, command/status registers, battery-backed CMOS memory, etc. Although the present invention is illustrated with an IBM-compatible type PC system, it is understood that the present invention is applicable to other types of computer systems and processors as known to those skilled in the art.

A library 121 of predetermined or written procedural objects or "gizmos" implemented according to the present invention is stored in the storage system 121. The memory system 106 stores data 130, program code 132 and an edit utility application 134 for execution by the CPU 104. The program code 132 and the edit utility application 134 may be loaded into the memory system 106 from the storage system 120 during operation. The data 130 may include portions loaded from the storage system 120, such as any one or more gizmos from the library 121, and new or updated data created by the program code 132 or the edit utility application 134 during operation. The program code 132 may include a plurality of components, procedure modules and/or procedural objects that a user combines to create intrinsically parallel program code. The computer system 100 is included to illustrate that an object model according to the present invention may be realized on a modern computing machine with a CPU, random access memory (RAM) and external storage capacity (ESC), such as the storage 120. The graphic user interface (GUI) forms of the model further require a display capable of displaying graphics. There are no explicit restrictions on CPU architecture or display technology.

Referring now to Figure 2, a block diagram is shown of a network system 200 that communicatively couples a plurality of computer systems or computing devices 202, 204, 206, 208, 210, etc. together via a communication medium 220. Any one or more of the computing devices 202-210 may be implemented in the same or a similar manner as the computer system 100. Of note, however, any one or more of the computing devices 202-210 may alternatively include only processing capabilities for purposes of distributed processing. The network system 200 may include any one or more network devices (not shown), such as hubs, switches, repeaters, bridges, routers, etc. The network system 200 may operate according to any network architecture, such as Ethernet™, Token Ring, etc., or combinations

of such architectures at any available speed, such as 10 Megabits per second (Mbps), 100 Mbps, 1 Gigabits per second (1Gbps), etc. The network 200 may form any type of Local Area Network (LAN) or Wide Area Network (WAN), and may comprise an intranet and be connected to the Internet.

The following is a description of a programmable object model. A central programming element is a procedural object called a "gizmo". A gizmo can be a stand-alone program, but may also be combined with other gizmos to extend their joint functionality. Further, combined gizmos can be encapsulated behind another gizmo object to define a new gizmo, where a new gizmo is created from composed gizmos.

In particular, new functionality is constructed by combining the components of a number of gizmos. Components are associated with input/output (I/O) and generally comprise values and events. This allows the newly defined composite functionality to be encapsulated into a new gizmo class. The gizmo class is defined as an interface into the composed functionality. Several key interface components are selected from among the components of combined gizmos. There are no components which are required to be in an interface. Every gizmo is created from a gizmo class. The gizmo or gizmo instance (copy of an underlying gizmo) is stored in a container. The composition within a container forms the basis for a new gizmo class.

Gizmo procedural objects as described herein are interesting and useful as a method for composing component-based programs. This could be used for network programming models or automatic program generation. The realization of the gizmo model into practice could be implemented without a graphic user interface (GUI). An application program that interfaced to a database of gizmos could activate, compose, abstract, and instantiate new gizmos. This makes it possible to use the gizmo model for automated programming applications.

However, gizmo programming becomes even more practical when manipulated via a GUI, and is well suited for user programs developed through a GUI. This introduces the additional requirements of appearance and user interaction. In order for a human user to interact with a gizmo they must be able to initiate the start events, recognize end events,

change input values, view changes to output values, and experience audio visual effects. How this is realized could vary greatly depending on the implementation of the GUI.

All GUIs accommodate some representation of the content of value components as text and graphics. However, the additional visual or audio capabilities of a GUI introduce the need for visual entry methods of value content. This is provided to the GUI environment with gizmos that relate a visual representation with its associated values. The visual element can be interactive and allow the user to change the value by manipulating the visual representation.

Figure 3 is a block diagram of a procedural object or gizmo 300 including associated and dependency relationships illustrated according to the Unified Modeling Language (UML). The gizmo 300 may include one or more components including zero or more ("1"--"0..*") start events 302, end events 304, and values 306. A component is either an event or a value. An event may be a start event, an input event, an output event or an end event. Each value or value component 306 may also comprise an input or output value and generally contains a single value that may be any one of various types, such as alphanumeric, symbolic, graphical, textual, etc. Activation of the start event 302 initiates or calls a procedure module 308 of the gizmo 300, which may access any input value components 306 (access), if any, modify output values 306 (modify), if any, and then activate any end event components 304. Of note, the gizmo 300 may have multiple start events and end events and be associated with multiple values.

Figure 4 is a block diagram illustrated in UML format of an exemplary activation sequence 400 of a procedural object or gizmo, such as the gizmo 300. A gizmo may respond to input values or events, modify output values and/or activate output or end events. Although values may be both an input value and an output value, values are preferably of a type that can be interpreted by the gizmo. In this way, the gizmo object language is defined as being strongly typed. The activation sequence 400 illustrates a user 402 that activates a start event 404. The start event 404 then calls a procedure or procedure module 406 that performs user-defined operations. As shown in Figure 4, the procedure module 406 accesses an input value 408, modifies an output value 410 and activates an end event 412. The value

access and modification and the event activation operations could be performed in any desired order and include numerous combinations of events and values.

Events are typically activated by a user, an application or an end event of another gizmo. For example, a user may define a procedure module that is to be activated when the user, through certain operations of an input device, such as the mouse 114 or keyboard 116, activates a start event. Event activation by the user may be accomplished when the user presses a button, via an input device, that correspondingly presses a graphical user interface (GUI) button on the display 112 that represents the event, or invokes a program incorporating the event. Alternatively, a separate program, application or utility may be used to activate a start event. When a start event is activated, numerous possibilities exist for the operations that are performed by the associated procedure modules. Although the above examples only demonstrate an event that activates a procedure module, an event that is activated by a procedure module, and an event that is activated by a user, the user could define alternative events such as an event that both activates and is activated by a procedure module.

Figure 5 is a block diagram in UML format of an exemplary combiner 500 that illustrates relationships between the combiner 500 and event components 502 or value components 504. A user creates new functionality in a gizmo by combining, linking or otherwise "stitching" events with events or values with values via the combiner 500. For example, the user may combine components from gizmo 300 with edit utility application 134 to create new functionality and a new gizmo. The combiner 500 allows events to be combined with events and values to be combined with values. When two or more components are combined, they become equivalent components. Activation of any one event causes activation of any events combined with the activated event. This is conceptually treated as concurrency, but this is not required. It is sufficient to say that the ordering of combined event activation is non-deterministic. Likewise, modification of any one value causes the same modification of any values combined with the modified value. The combiner 500 is a program module or code that creates a separate combiner instance between combined events or values, as further described below.

Figure 6 is a block diagram in UML format of an exemplary activation sequence 600 when the combiner 500 is used to combine multiple event components and associate a

procedure module. Using the combiner 500, a combiner instance 610 is created that combines a start event component 602 (Event0) with two other event components 614 (Event1) and 616 (Event2). A parallel fork 612 indicates association of the combiner instance 610 with both of the event components 614, 616, which reflects the combination of the event components 602, 614 and 616. Another parallel fork 608 indicates association of the event component 602 with both the combiner instance 610 and a procedure module 618 (Procedure1). As illustrated, the start event component 602 is activated either by a user 604 or a procedure 606 (Procedure0). When the start event component 602 is activated, the combiner instance 610 is invoked in parallel with, or effectively at the same time as, the procedure module 618 as indicated by the parallel fork 608. Also, the event components 614, 616 are both activated in parallel by the combiner instance 610 as indicated by the parallel fork 612. As a result, the procedure module 618 is invoked in parallel with activation of the event components 614 and 616 in response to activation of the start event component 602.

Figure 7 is a block diagram of an exemplary modification sequence 700 when the combiner 500 is used to combine multiple value components. Using the combiner 500, a combiner instance 708 is created that combines a value component 702 (Value0) with two other value components 712 (Value1) and 714 (Value2). A parallel fork 710 indicates association of the combiner instance 708 with both of the value components 712 and 714, which reflects the combination of the value components 702, 712, and 714. As illustrated, the first value component 702 is modified either by a user 704 or by a procedure 706, which invokes the combiner instance 708. The combiner instance 708 modifies both of the value components 712 and 714 to reflect the same modification of the value component 702. Effectively, for the modification sequence 700, modification of the value component 702 causes the value components 712 and 714 to be modified in the same way at effectively the same time as indicated by the parallel fork 710. Of course, the user may use the combiner 500 to combine any number of values or value components so that all combined values become equivalent to each other. Also, although value components of any type may be combined, only values of the same type will have the same value. This is conceptually treated as the same storage space, and so is implicitly concurrent.

Figure 6 and Figure 7 each illustrate an example of the intrinsically parallel nature of the programming system according to the present invention. When the combiner 500 is used

to combine events, these combined events are essentially activated concurrently when one of the events is activated. Likewise, when the combiner 500 is used to combine values, these combined values are essentially modified concurrently when one of the values is modified. A plurality of gizmos or procedural objects, each including one or more value and/or event components, are linked together using a combiner, such as the combiner 500. In this manner, multiple events, values, procedures (or procedural modules) of multiple procedural objects may be activated or modified, as the case may be, effectively at the same time. A programmer or user may force sequential operation when useful or desired.

Figure 8 is a block diagram in UML format of an exemplary gizmo class 800 illustrating gizmo relationships between zero or more (0..*) value components 802 and zero or more event components 804. A new gizmo may be created from previously composed gizmos, such as gizmos stored in the library 121. The gizmo class 800 allows newly defined composite functionality to be encapsulated into a new gizmo or gizmo class. The gizmo class 800 is defined as an interface into the composed functionality. Several key interface components are selected from among the components of combined gizmos, although there are no components that are required to be an interface. For example, an interface may consist of only a value component and no events, where the gizmo might access some external entity such as a system clock or monitor to available memory. The interface might also contain only events which play sounds. The gizmo class 800 is essentially a gizmo including a higher level of abstraction than gizmo 300.

Figure 9 is a block diagram in UML format of an exemplary gizmo container 900 that is used to store gizmo instances that are accessible for use in constructing a gizmo class 800. Essentially, every gizmo, such as the gizmo 902, is created from a gizmo class, such as the gizmo class 904. The gizmo or gizmo instance 902 is stored in the gizmo container 900. Only gizmos within the same container may be composed. The composition within the container 900 forms the basis for a new gizmo class. A gizmo class cannot be directly or indirectly instantiated into a container with which it is defined because doing so would create an illegal recursive gizmo class definition.

Figure 10 is a block diagram in UML format illustrating steps of a gizmo instantiation sequence 1000. A user 1002 initiates a make gizmo procedure to create a gizmo class 1006.

The gizmo class 1006 is then added to a container 1008. In this manner, through means such as the edit utility application 134, gizmo instances are made available for user access to create further gizmo classes.

Exemplary embodiments will now be described in which a graphical user interface (GUI) is used to represent gizmos or procedural objects having components, i.e., values and events, and relationships among the gizmos. Although the GUI can operate in many different environments such as World Wide Web (WWW) environments, custom GUI environments, Windows ®, etc., the exemplary embodiment will be described as it would operate in a Windows ® environment.

Figures 11A and 11B are illustrations of an exemplary graphical representation of a procedural object or a gizmo instance or simply a gizmo 1100 as displayed in a GUI environment. A display, such as the display 112, shows the "face" of one or more gizmos, such as the gizmo 1100, where the gizmo face displays one or more of the included value and event components. The gizmo typically includes one or more underlying procedural modules associated with the components on the gizmo face. Although the underlying procedural module is not shown, its function is often alluded to by the name assigned to the gizmo and/or the symbols used for event buttons. The graphical representation of the gizmo 1100 includes a graphical representation of a button widget or simply a button 1102 having a "+" label. Labels generally have no meaning in the gizmo other than a symbolic meaning to the user indicating the underlying function. For example, the "+" label indicates an arithmetic add function, although the user may amend the functionality as further described below. The user may also edit labels as desired. The button 1102 represents an underlying input or start event that a user activates by pressing the button 1102 via an input device, such as, for example, pressing the button 1102 via standard mouse operations using the mouse 114 and the display 112. The buttons of a gizmo are often referred to as input, start, output or end event buttons.

The gizmo 1100 also includes two input value widgets or boxes 1104 and 1106, an output value box 1108, and two end event buttons 1110 (labeled "Done") and 1114 (labeled with an exclamation point "!"). The user may enter alphanumeric values into the value boxes 1104, 1106 via input devices, such as the mouse 114 and keyboard 116. As shown in Figure

11A, the user has entered the numeric value 123.45 into the value box 1104 and the numeric value 67 into the value box 1106. As illustrated in Figure 11B, when the start event button 1102 is pressed by the user, the underlying addition procedure is performed on the two input values and the input values are added. The result of the addition, or the number 190.45, is an output value placed by the underlying procedure into the value box 1108 as shown in Figure 11B. Further, once the addition has completed, a dotted line 1112 representing highlighting appears around the end event button 1110. The dotted line 1112 representing highlighting indicates to the user that the end event has been activated since the addition procedure has successfully completed. If the addition fails for any reason, the end event button 1114 is highlighted (not shown) indicating that an error end event occurred. For example, an error is indicated if an input value was not a valid type, such as not a number.

Although the graphical representations of buttons 1102, 1110, and 1114 are used to communicate corresponding underlying events to the user, other graphical representations are possible to display the underlying events. For example, color changes, blinking graphics, modified graphics, new graphics, brightness changes, symbol changes and/or additions, etc. may be used to indicate activation of or a status change of an event. As described more fully below, the events underlying buttons 1102, 1110, and 1114 can be alternatively activated by other associated underlying events, such as activation of combined events from other gizmos.

Figures 12A, 12B and 12C are exemplary illustrations of the graphical user interface representation of the gizmo instance 1100 during a "stitching" operation. A stitch 1200 is represented by a solid graphical line that is created by the user via an input device such as the mouse 114 and/or the keyboard 116. For example, using the mouse 114, a valid stitch is created when the a cursor 1202 is dragged from one value box to another, or when it is dragged from one event button to another event button, while a button on the mouse is pressed. For example, the stitch 1200 results when the user presses a mouse button of the mouse 114 while the cursor 1202 is located on the value box 1106, while the cursor 1202 is dragged via the mouse 114 to the value box 1108 while the mouse button down is held down, and after the mouse button is released when the cursor is located on the value box 1108. The same stitch 1200 may be made in the opposite direction, such as "dragging" the cursor 1202 from the value box 1108 to the value box 1106. When the stitch 1200 is completed between the value boxes 1106 and 1108, the value boxes 1106 and 1108 are combined and the

underlying values become equivalent. In this manner, whenever a value in either of the value boxes 1106 or 1108 is updated, the update is reflected in the other of the value boxes 1106, 1108. Of course, a similar combination occurs when event buttons are combined through stitching.

Figure 12B is an illustration of the exemplary GUI representation of the gizmo instance 1100 after completion of the stitching operation of Figure 12A. As stated, value boxes 1106 and 1108 are now stitched together as represented by the dotted lines 1204. Of note, when the mouse pointer 1202 is placed over the value box 1108 in an edit mode, both of the value boxes 1106 and 1108 are highlighted as represented by the dotted lines 1204 around the stitched value boxes 1106 and 1108. As indicated previously, highlighting may be represented in any desired manner to convey stitched components to the user, such as color changes, blinking graphics, modified graphics, new graphics, brightness changes, symbol changes and/or additions, etc.

The consequence of stitching the value boxes 1106 and 1108 together is that the gizmo instance 1100 is converted from an adder into an incrementor. When the user presses the start event button 1102 as shown in Figure 12B, the values in the value boxes 1104 and 1106 are added together and the resulting output value, i.e., 190.45, is placed into the value box 1108 as shown in Figure 12C. In addition, an underlying combiner instance (not shown) also updates the value in the value box 1106 to reflect the new value 190.45 placed in the value box 1108. In this manner, the value in the value box 1108 is "incremented" by the value in the value box 1104 every time the button 1102 is pressed. It is noted that had the value boxes 1104 and 1108 been stitched together instead, the resulting incrementor would be similar except that the value in the value box 1108 would be incremented by the value in the value box 1106 every time the button 1102 was pressed.

Figure 13 is an illustration of an exemplary play space 1300 that illustrates gizmo composition by using existing gizmos, such as from the library 121, to create a new gizmo with new functionality. In this case, the user creates and edits an abstraction interface called a gizmo face 1302 for the play space 1300. A play space may have more than one gizmo face, although only one is shown for purposes of clarity and simplicity. The play space 1300 can be saved into a file and reopened for later editing. It can also be loaded as a new class of

gizmo that can be instantiated into some other play space, thus creating a hierarchy of user defined gizmos.

Although not shown, the gizmo face 1302 is initially blank and thus includes no components (no values or events). The user calls and enters instances of an existing multiply gizmo from a memory of existing gizmos, such as from the library 121, resulting in two multiply gizmo instances 1304 and 1306. The user further calls and enters two instances of an existing divide gizmo from the library 121 resulting in two divide instances 1308 and 1310. The divide gizmos 1308 and 1310 are used for purposes of creating conversion factors between degrees and radians. In particular, the user enters the value 180 (degrees) into a first input value box 1332 and the value 3.141592 (or pi, i.e., π, radians) into a second input value box 1334 of the divide gizmo 1308. The user then presses a start event button 1335 (labeled with a divide symbol "/") of the divide gizmo 1308, where the underlying divide procedure divides 180 by π to calculate the output value 57.29565 degrees/radians (radian to degree conversion factor) and places the output value in an output value box 1340 of the divide gizmo 1308. In a similar manner, the user enters the value π (radians) into a first input value box 1336 and the value 180 (degrees) into a second input value box 1338 of the second divide gizmo 1310. The user then presses a start event button 1339 (also labeled "/") of the gizmo 1310, where the underlying divide procedure divides π by 180 to calculate the output value 0.01745 radians/degrees (degree to radian conversion factor) and places the output value in an output value box 1342.

The output value box 1340 is stitched to an input value box 1326 of the first multiply gizmo 1304 (either before or after the first divide function is performed) via a stitch 1301. Likewise, the output value box 1342 is stitched to an input value box 1330 of the second multiply gizmo box 1306 (either before or after the second divide function is performed) via a stitch 1303. In this manner, the conversion factors in the respective value boxes 1340 and 1342 are copied to the respective value boxes 1326 and 1330. An input value box 1324 of the multiply gizmo 1304 is stitched to an output value box 1344 of the multiply gizmo 1306 via a stitch 1305 and an input value box 1328 of the multiply gizmo 1306 is stitched to an output value box 1346 of the multiply gizmo 1304 via a stitch 1307. The gizmo 1306 also has an end event button 1350 (labeled "Done") and an error event button 1354 (labeled "!"), which

are both stitched to corresponding end event buttons 1348 (labeled "Done") and 1352 (labeled "!"), respectively, of the multiply gizmo 1304 via respective stitches 1309 and 1311.

The user also creates new value boxes and event buttons in the gizmo face 1302 by creating interfaces between existing value boxes and event buttons of the multiply gizmos 1304 and 1306. An interface is created by copying a value or event from a gizmo to the gizmo face 1302, thereby creating a new value or event that is automatically stitched to the interfacing value or event. In particular, the user copies or interfaces the input value box 1324 of the multiply gizmo 1304 to create a value box 1320 in the gizmo face 1302 via an interface 1321. Likewise, the user copies or interfaces the input value box 1328 of the multiply gizmo 1306 to create a value box 1322 in the gizmo face 1302 via an interface 1323. In a similar manner, the user copies or interfaces an input event button 1356 (labeled with a multiply symbol "X") of the multiply gizmo to the gizmo face 1302 to create a conversion event button 1314 via an interface 1325. The event buttons 1356 and 1314 are thus stitched together. Likewise, the user copies or interfaces an input event button 1358 (also labeled with "X") of the multiply gizmo 1306 to create a conversion event button 1312 of the gizmo face 1302 via an interface 1327. The event buttons 1358 and 1312 are likewise stitched together. The event buttons 1312 and 1314 are modified and re-labeled to denote new or modified functionality of the gizmo face 1302. In particular, the event button 1312 is labeled "radians" to denote conversion from degrees to radians and the event button 1314 is labeled "degrees" to denote conversion from radians to degrees. The end event button 1348 and the error event button 1352 of the multiply gizmo 1304 are both interfaced to the gizmo face 1302 to create an end event button 1318 (also labeled "Done") and an error event button 1316 (also labeled "!"), respectively, for the gizmo face 1302 via respective interfaces 1329 and 1331. In this manner, the exemplary play space 1300 is used to compose a new gizmo face 1302 to form a two-way radian and degrees converter gizmo.

In operation, the user places a value representing degrees in the value box 1322 of the gizmo face 1302 and presses the radians start event button 1312. When the degrees value is placed in the value box 1322, the degrees value in the value box 1322 is reflected into the value box 1328 of the multiply gizmo 1306 and also in the value box 1346 of the multiply gizmo 1304. When the radians event button 1312 is pressed, the multiply event button 1358 is also activated due to the stitching therebetween, causing the multiply gizmo 1306 to

multiply the input degrees value in the value box 1328 by the degrees to radians conversion factor in the value box 1330 resulting in a corresponding radians value of 0.5934 to be placed into the output value box 1344 of the multiply gizmo 1306. The radians value of 0.5934 is also copied to the value box 1324 of the multiply gizmo 1304 and thus also into the value box 1320 of the gizmo face 1302. The underlying multiply procedure of the multiply gizmo 1306 also highlights the output event button 1350 (by activating the underlying output event), which causes activation and highlighting of the output event buttons 1348 and 1318 due to event stitching. In this manner, the user entered a degrees value in the degrees value box 1322, pressed the radians event button 1312, and gets the output radians value in the value box 1320 and sees the end event button 1318 highlighted indicating completion of the conversion.

In a similar manner, the user may place a radians value in the value box 1320 of the gizmo face 1302 and then press the degrees event button 1314. The radians value in the value box 1320 is reflected into the value box 1324 of the multiply gizmo 1304 and also into the value box 1344 of the multiply gizmo 1306. When the degrees event button 1314 is pressed, the multiply event button 1356 is also activated, causing the multiply gizmo 1304 to multiply the input radians value in the value box 1324 by the radian to degree conversion factor in the value box 1326 resulting in a corresponding degree value of 34 to be placed into the output value box 1346 of the multiply gizmo 1304. The degree value of 34 is also copied to the value box 1328 of the multiply gizmo 1306 and thus also into the value box 1322 of the gizmo face 1302. The underlying multiply procedure of the multiply gizmo 1304 also activates the output event button 1348, which causes activation of the output event buttons 1350 and 1318 to also be activated due to event stitching. In this manner, the user entered a radians value in the radians value box 1320, pressed the degrees event button 1314, and got the output degrees value in the value box 1322 and sees the end event button 1318 highlighted indicating completion of the conversion.

The play space 1300 demonstrates how value boxes may be used to represent both input values and output values for the same gizmo and how event buttons can represent start events or end events, as well as being user- or procedure-activated events. The radian/degree conversion functionality of the gizmo face 1302 is made possible by stitches between the values and events of the gizmo instances 1304, 1306, 1308, and 1310, and the stitches to the

gizmo face 1302. In summary, Figure 13 illustrates one example of relationships that can be created between gizmos according to the principles of the present invention. The gizmo face 1302, which is depicted by the relationships that were defined with the gizmo instances 1304, 1306, 1308, and 1310, can now be used as a gizmo instance or building block for future gizmo faces. Of particular note, the play space 1300 can be saved into a file and reopened for later use and/or editing. The gizmo instances 1308 and 1310 are stored in the play space 1300 with the radian to degree conversion factors and degree to radian conversion factors, respectively, remaining unchanged. The gizmo with the gizmo face 1302 can also be loaded as a new class of gizmo that can be placed into some other play space, thus creating what appears to be a hierarchy of user-defined gizmo instances. However, although the gizmo definition relationships are hierarchical, the gizmos execute concurrently. These numerous relationships, i.e., stitches, that are defined between gizmo instances are often referred to as a "fabric".

The example of Figure 13 illustrates the intrinsically parallel nature of the programming system of the present invention. This intrinsically parallel nature is particularly suited for operation of programs in a distributed processing network, such as the network of Figure 2, by assigning individual gizmos or groups of gizmos to different processors in the network so that the gizmo functionality of one gizmo is processed separately from the gizmo processing performed in other processors. The stitches, however, are still in effect across a network to enable the user to force sequential processing where and when desired. In effect, the fabric created between gizmos, even when distributed across a network to multiple computers or processors, maintain the same functionality as though the program were executed on a single computer.

Further, any existing or new gizmo can be operated through an interpreter or a compiler. An interpreter allows the gizmo face 1302 to operate according to the basic functionality of the gizmo by making complete copies of the gizmo hierarchy for each gizmo instance and ignoring any inefficiency created by duplicate code in the copies. This is particularly advantageous for a user creating a program in gizmo format and testing the program on the fly in an edit mode or the like. Once the user is satisfied with the functionality, the new gizmo may be stored away, such as in the library 121, for later use or use in creating other gizmos. A compiler may be used to optimize the code in any gizmo to

maximize the efficiency of the program code. In general, a compiler reduces or otherwise eliminates any redundant or duplicate code, optimizes portions of the code and simplifies any code where necessary to achieve optimal program code.

Figure 14 is an illustration of another play space 1400 according to an embodiment of the present invention to illustrate stitches between gizmo instances to create or "program" new functionality. The play space 1300 is saved into a file as a new "convert" gizmo and the user retrieves an instance of the convert gizmo 1402 into the play space 1400. The convert gizmo 1402 includes the underlying radians/degrees functionality previously programmed by the user via the play space 1300. The user may retrieve as many instances of the convert gizmo 1402 as desired into the new play space 1400. The user also retrieves another gizmo instance 1404, which is another instance of the adder gizmo, the same as the gizmo instance 1100. The user has stitched an input value box 1410 with an output value box 1408 creating an incrementor as previously described and as illustrated by a stitch labeled "B". The user enters the value 45 into the input box 1410 and into another input box 1405 of the incrementor gizmo 1404 and presses an add start event button 1406. The output value of 90 appears in the output value box 1408 and is reflected back into the input value box 1410 as a result of the stitch B. Meanwhile, the user has also stitched the output value box 1408 with an output value box 1411 of the gizmo 1402 as illustrated by a stitch labeled "A", and has stitched a normal end event 1412 of the gizmo 1404 with a radians button 1414 of the gizmo 1402 as illustrated by a stitch labeled "C".

In this manner, The two gizmo instances 1402, 1404 are stitched via stitches A, B and C to implement a radian conversion iterator that sequences through radian values based on a degree increment, which is 45 degrees in the case shown. When the button 1406 is pressed, the underlying function of the gizmo instance 1404 increments the value in the output value box 1408 by the input value in the value box 1405 and activates the end event 1412 as before. Further, the value in the value box 1408 is copied to the value box 1411 of the gizmo 1402 and the start event 1414 is activated by the activation of the end event 1412. The value in the value box 1411 from the gizmo 1404 represents degrees that is converted by the gizmo 1402 to radians resulting in a radians value of 1.5708 in a value box 1415, and an end event 1417 (labeled "Done") is then highlighted (not shown). A subsequent pressing of the button 1406 causes an increment of 45 degrees to the value box 1411, which is converted to a

corresponding increment of the radians value in the value box 1415. This example illustrates how a play space can be used as a test bench for newly defined gizmo instances. The natural visualization capabilities make GUI-based programming appealing.

Figure 14 illustrates how sequential behavior is introduced into the concurrent behavior of combined events. Many gizmo instances provide end events which allow sequential dependencies to be established between gizmo instances. The user has the option to continue this approach with newly created gizmo classes. A gizmo instance, which is intended as a building block for other gizmo instances, would be a good example of when an end event should be included. However, gizmo instances which are expected to be a finished product, or which run continuously, do not need to include end events. Of note, gizmo instances represented by a GUI can be copied into the same or a different play space. When multiple gizmo instances are copied at the same time, their shared stitches are also copied. These capabilities enable the user to become a programmer and to compose or program new functionality easily and quickly. Further, the programming code is intrinsically parallel. The graphical gizmo programmer can edit the play space where the gizmo instance is defined through a menu on the play space. This edit definition capability can greatly improve the effectiveness of the development environment.

Figures 15 and 16 are graphical illustrations of editing modes available in a programming system according to the present invention. As shown in Figure 15, the play space 1400 is illustrated in an edit mode and when the mouse 114 is used to place a cursor 1506 over the value box 1408 of the gizmo 1404, dashed lines 1500, 1502, and 1504 surrounding the value boxes 1408, 1410, and 1411, respectively, are displayed to represent highlighting of the stitched values. All value boxes that are stitched together are highlighted when the cursor 1506 is positioned on top of any one of the stitched value boxes. For ease of understanding the exemplary editing functionality, the highlighting has been labeled GREEN because, initially, all highlighting is in green.

In Figure 16, the user clicked a button on the mouse 114 while the cursor was positioned over the value box 1408 to "unstitch" the value box 1408 from the value boxes 1410 and 1411. As a result, the dashed line 1500 representing the highlighting around the value box 1408 changed from green to red as indicated by label RED. The value boxes 1410

and 1411 are still stitched as indicated by the dashed lines 1502 and 1504 still labeled GREEN. Thus, the value box 1408 is removed from the original fabric of stitches A and B, while the value boxes 1410 and 1411 remain in the fabric. In this manner, mouse operations allow selected value widgets from the same or different gizmo instances to be removed from or added to their original fabric according to the programmer's choices during an edit mode.

Figure 17 shows the play space 1400 to illustrate the gizmo instances 1402, 1404 with a new stitch, represented by a stitch labeled "D", that was created during the editing of the gizmo instances 1402, 1404. The stitch C between value boxes 1410 and 1411 is unchanged. Frequent changes to the stitches represented in a play space may be made as desired when programming new gizmo functionality, and the interactive fabric editing is designed to expedite changes of any degree of complexity. As stated, gizmo instances can have stitches between underlying values either added to them or removed from them regardless of whether the stitch is within the boundaries of the gizmo instance or extends to other gizmo instances. Further, in this exemplary embodiment, stitches between events are edited in the same manner.

Figure 18 is an illustration of a play space 1800 according to an embodiment of the present invention to illustrate exemplary stitches between gizmo instances to create or "program" iterative functionality in the play space 1800. Iteration is the basis for many useful programs and gizmos can be caused to iterate by being stitched into a loop. A loop is created by stitching end events to start events, thereby creating a serial dependency between gizmos. A series of events may contain a loop if one of the events, which is later in the series, is stitched to an earlier event in the series. This creates a cycle in the event sequence and causes the series to restart itself, i.e., to cycle or iterate. As shown in Figure 18, a minimalist loop is created by stitching together an end event and a start event of the same gizmo, i.e., gizmo 1802. This causes the gizmo 1802 to restart itself each time it has finished, thus resulting in a gizmo which runs continually and creating a gizmo that is advantageous to parallel programming.

Although not shown in Figure 18, the play space 1800 is initially blank and thus includes no gizmo instances. The user calls and enters instances of existing gizmos from the library 121 of existing gizmos, resulting in the user retrieving an instance of a delay gizmo

1802, and an instance of an add gizmo 1804 into the play space 1800. Of note, the add gizmo 1804 includes the underlying addition functionality previously described and the gizmo instance 1802 includes the underlying functionality of a delay procedure. In gizmo instance 1802, the user has stitched a start event button 1806 (labeled "Go") with an end event button 1808 (labeled "Done") as illustrated by a stitch 1810. The stitch 1810 creates a continuous loop in the gizmo instance 1802 because its start event is stitched to its end event. However, because gizmo instance 1802 is a delay gizmo, each iteration of the loop is delayed by the amount of time designated in value box 1812. The delay value in the value box 1812 may have any time units, such seconds, minutes, hours, days, etc. and is selected as seconds in the embodiment shown.

Also in the play space 1800, a stitch 1814 is illustrated that the user created between the end event button 1808 of the delay gizmo 1802 and a start event button 1816 of the gizmo instance 1804. Thus, each time the end event 1808 is activated, both the start event 1806 and the start event 1816 are activated. As previously described, the start event button 1806 activates a delay, concurrently, the start event 1816 activates an addition of the numeric values in each input value box 1818 and 1820. In addition, as shown by a stitch 1817, the user has stitched input value box 1818 to output value box 1822 to create an incrementor similar to the incrementor of Figure 12C. Thus, the value in output value box 1822 is incremented by the value in input value box 1820 each time the loop of the gizmo instance 1802 iterates, i.e., every five (5) seconds as indicated by the value 5 in the value box 1812. If, during the five second delay period, any of the event buttons 1806, 1808, or 1816 were somehow activated externally to the loop, the gizmo instance 1804 increments and the delay gizmo 1802 begins counting a new delay period for the interation. This occurs because the event buttons 1806, 1808, and 1816 are stitched together and are all activated together (and thus, the start event 1806 is re-activated regardless of completion of the delay) when any one of them is activated. Of course, the delay period illustrated in value box 1812 could be set to a different value and the delay also depends on the underlying procedure that is run when the start event of the gizmo instance 1802 is activated.

A continuous/infinite loop such as the loop created by operation of the gizmo instances of play space 1800 can be broken by using events from a conditional gizmo. A conditional gizmo is created when a gizmo has more than one end event that results from the

same start event. Although a conditional gizmo has multiple end events, only one of the end events is activated upon completion of an underlying gizmo procedure and as determined by the underlying gizmo procedure. There are many possible conditional gizmos. For example, every arithmetic gizmo is conditional because, as well as the normal "Done" end event labeled, they may also activate an error "!" end event if a non-number is provided as input. Exemplary conditional gizmos are gizmos that perform comparisons between values, like Compare Number and Compare String.

Figures 19A - 19D are illustrations of an exemplary play space 1900 that illustrate an example of iterative/conditional gizmo composition by using existing gizmos to create a new gizmo with new functionality. The user programs a gizmo using the play space 1900 to calculate the square root of the input value using an algorithm, such as, for example, Newton's square root loop algorithm. Newton's square root loop algorithm may be expressed by the following equation 1:

$$x(i) = (Y/x(i-1) + x(i-1))/2;\ x(0) = Y/2 \qquad (1)$$

where x, Y and epsilon "ε" are real values, "i" is an integer incremented beginning with one (1) until a termination criterion is met, and the termination criterion equals $|Y\text{-square}(x(i))| < \varepsilon$. The value epsilon "ε" is an error value that may be simulated as an error difference between the last two estimates rather than squaring the last estimate and comparing with the input value. The gizmo is programmed to perform Newton's square root loop algorithm as an iterative process that converges on the solution and that completes when a prior estimated value and the current estimated value are the same up to a predetermined decimal place.

The user starts by opening a new play space 1900 and entering a blank gizmo face 1902 as shown in Figure 19A. An input value box 1918 is added to the gizmo face 1902 to enable a user to enter a numeric input value, a start event 1922 labeled "Go" is added to enable the user to activate the programmed procedure, an output value box 1920 is added to display the square root of the numeric input value as calculated by the underlying procedure, an end event 1924 labeled "Done" is added to indicate proper completion of the procedure and an error end event 1926 labeled "!" is added to detect an error of the procedure. The user then retrieves from the library 121 a sign gizmo 1914 that ensures the input value Y is

positive, a divide gizmo 1904 to perform an initial divide by 2 to generate an initial value (Y/2) for the iterative process, another divide gizmo instance 1906 to divide the input value by the previous estimate (Y/x(i-1)), an average gizmo 1916 to add the result from the divide gizmo 1906 with the previous estimate and divide by two (by taking the average), and a compare gizmo 1912 to compare the latest two estimates to a predetermined decimal point. In the embodiment shown, the result is considered accurate if within seven significant digits. A couple of copy number gizmo instances 1908 and 1910 are added to control the iterative process as further described below.

With reference to Figures 19B and 19C, the user stitches gizmo instances 1902, 1904, 1906, 1908, 1910, 1912, 1914, and 1916 in a manner that creates an underlying procedure that performs Newton's square root algorithm on an input value entered into an input value box 1918. In particular, stitches 1901 and 1903 are placed to stitch the input value box 1918 with a numerator input box 1944 of the divide gizmo 1904 and an input value box 1990 of the sign gizmo 1914. Stitches 1905 and 1907 are placed to stitch the input value box 1918 with an input value box 1941 of the copy number gizmo 1908 and a numerator input box 1952 of the divide gizmo 1906. The start event button 1922 is stitched to a start event button 1930 of the sign gizmo 1914 with stitch 1909. The sign gizmo 1914 includes a negative output value event 1934 and a zero output value event 1936, which are both stitched to an error event button 1932 via stitches 1911 and 1913, respectively. The zero event button 1932 is also stitched to the error event button 1926 of the gizmo face 1902. In this manner, an error is indicated if the input value, copied to the input value box 1990, is zero or negative. Of course, the square root of zero is zero, but, in this case, the algorithm fails.

A positive output event button 1938 of the sign gizmo 1914 is stitched to a start event button 1942 of the copy number gizmo 1908 and a divide start button 1940 of the divide gizmo 1904. An output value box 1943 of the copy number gizmo 1908 is stitched to the output value box 1920, an output value box 1984 of the copy number gizmo 1910 and an input value box 1970 of the compare numbers gizmo 1912 via stitches 1921, 1923 and 1925, respectively. An output value box of the divide gizmo 1904 is stitched to another input value box 1972 of the compare numbers gizmo 1912, to an input value box 1982 of the copy number gizmo 1910, to an output value box 1964 and an input value box 1960 of the average gizmo 1916 and to a denominator input box 1954 of the divide gizmo 1906 via stitches 1927,

1929, 1931, 1933 and 1937, respectively. An output event button 1948 of the divide gizmo 1904 is stitched to a divide start event button 1950 of the divide gizmo 1906 via a stitch 1939. An output value box 1956 of the divide gizmo 1906 is stitched to an input value box 1962 of the average gizmo 1916 via a stitch 1941. An output event button 1958 of the divide gizmo 1906 is stitched to start event button (labeled "Ave") 1959 of the average gizmo 1916 via a stitch 1943. An end event button 1966 (labeled "Done") of the average gizmo 1916 is stitched to an input event button 1968 of the compare numbers gizmo 1912 via a stitch 1945. Less than "<" and greater than ">" output event buttons 1976 and 1978, respectively, are stitched together via stitch 1947 and to an input event button 1980 (labeled "Go") of the copy number gizmo 1910 via stitch 1949. An output event button 1986 (labeled "Done") of the copy number gizmo 1910 is stitched to the divide input event button 1950 of the divide gizmo 1906 via stitch 1951 to complete the iterative loop connection. Finally, an equal output event button 1974 (labeled "=") of the compare numbers gizmo 1912 is stitched to the output event button 1924 of the gizmo face 1902 via stitch 1953.

In operation, the user enters a value, in this case 20 (Figure 19B), into the input value box 1918 via an input device such as keyboard 116. The user then presses the start event button 1922, via a mouse operation using the mouse 114. The underlying procedure of the gizmo face 1902 then presents, in the output value box 1920 of the gizmo face 1902, a value, 4.472136 (shown in Figure 19D), representing the square root of 20 valid to seven significant digits and the end event button 1924 is highlighted. Figure 19B illustrates the play space 1900 prior to the user activating the start event button 1922, but after the user has added components to the gizmo face 1902, placed all the necessary stitches (necessary to perform Newton's square root algorithm) between the components of the gizmo instances in the play space 1900, and entered a value (20) into the input value box 1918 of the gizmo face 1902 and a value two (2.0) into the input value box 1928 of the divide gizmo instance 1904. Of note, the value 2.0 entered into the input value box 1928 is a constant that is not stitched and otherwise not effected and thus remains stored even after the play space 1900 is stored. This constant value remains stored and valid even when instances of the gizmo face 1902 are invoked into other play spaces. As illustrated in Figure 19B, the value (20) entered into input value box 1918 is copied to every value box to which the value has been stitched.

Figure 19C illustrates values in the value boxes of the gizmo instances 1902-1916 of the play space 1900 after the user has pressed the start event button 1922 and a first iteration of the gizmo instances 1902-1916 of the play space 1900 has occurred. It should be noted that after the user presses the start event button 1922, a start event button 1930 from the sign gizmo instance 1914 is activated to determine whether the value stored in input value box 1918 is a valid value. In this case, the value stored in input value box 1918 is valid -- i.e., positive number 20. Thus, if the value in input value box 1918 were non-numeric or anything but a positive number, the error end event button 1926 in the gizmo face 1902 is activated and the iteration procedure is terminated. However, because the input value is positive, a positive sign end event button 1938 is activated which, in turn, activates all events to which it has been stitched. Upon activation of the end event button 1938, the start event button 1942 of the copy number gizmo instance 1908 is activated and the value of input value box 1941, i.e., 20, is copied to output value box 1943, coincidentally, the identical value as assumed earlier. Of course, as conveniently illustrated, all stitched value boxes are updated with the most recent value of the output value box 1943.

The result of the activation of the start event button 1940 of the divide gizmo instance 1904 is that a divide is performed in the divide gizmo instance 1904 and the value in input value box 1944, i.e., 20, is divided by the value in input value box 1928, i.e., 2.0, to produce a value (10.0) in output value box 1946. The values in all value boxes that are stitched to the output value box 1946 are then updated with the new value (10.0). The end event button 1948 is then activated which, in turn, activates the start event button 1950 of the divide gizmo instance 1906.

The divide gizmo instance 1906, performs a divide, and divides the value 20 in the input value box 1952 by the value in the input value box 1954 resulting in the value 2.0 in the output value box 1956. Of course, all stitched values are updated and reflect the change in their respective value boxes. Advantageously, at this early point in the iterations of the underlying procedure of the gizmo face 1902, because the gizmo instances have been stitched in such a manner as to automatically initialize all value boxes of the play space, all value boxes have been assigned a value and initialization of the value boxes has been properly accounted for regardless of the input value entered into input value box 1918. Upon completion of the divide procedure of the divide gizmo instance 1906, the end event button

1958 is activated which, in turn, activates the start event button 1959 of the average gizmo 1916. Also, the value 10.0 from the value box 1954 is copied into the input value box 1960 and the value 2.0 from the value box 1956 is copied into the value box 1962.

The average gizmo instance 1916 performs an average operation on the two input values 10.0 and 2.0 in the input value boxes 1960 and 1962 to obtain the new output value, 6.0 (not shown) placed into the output value box 1964. The boxes stitched to value box 1964 are updated; however, these updates are not reflected in Figure 19C. Upon completion of the average procedure of the average gizmo 1916, the end event button 1966 is activated which, in turn, activates the start event button 1968 of the compare numbers gizmo 1912.

The compare numbers gizmo 1912 calls an underlying procedure in which the values 20 and 10.0 in input value boxes 1970 and 1972, respectively are compared. In this embodiment, if the values are identical up to seven significant digits, the end event 1974 is activated which activates the end event 1924 of the gizmo face 1902. In this case, ε is reached when the two values of the input value boxes 1970 and 1972 are identical up to seven significant digits. Of course, it is understood that the user could select ε to be any value, and the iteration example of Figures 19A-19D is exemplary for purposes of illustration. Nonetheless, in this first iteration of the gizmo instances in the gizmo face 1900, end event 1978 is activated (and end event 1976 is activated via stitch 1947) because the two values of the input value boxes 1970 and 1972 are not identical up to seven significant digits. Thus, a new iteration is begun where the start event button 1980 is activated in the copy number gizmo instance 1910. The value in input value box 1982 is copied into output value box 1984 and the value in output value box 1984, along with all stitched values, is updated to be the value 10.0 (not shown). After the copy, the end event button 1986 is activated which, in turn, activates the start event button 1950 of the divide gizmo instance 1906 and the iteration cycle continues with the new values in the value components of the gizmo instances of the play space 1900.

This iteration cycle described in relation to Figures 19A-19C is continued, and the values of the value component boxes are continuously updated, until the values of input value boxes 1970 and 1972 of the compare numbers gizmo 1912 are identical up to seven significant digits. At that point, the compare numbers gizmo 1912 activates the equal end

event button 1974 and the end event button 1924 of the gizmo face 1902 is activated to terminate the iteration cycle.

Figure 19D illustrates the values contained in each of the value boxes of the play space 1900 upon termination of the iteration cycle described above. As illustrated, the value in the input value box 1918 is 20. The value in the output value box 1920 of the gizmo face 1902 is 4.472136, i.e., the square root of 20 calculated up to seven significant digits. Of note, the value 4.472136 also appears in value boxes 1943, 1946, 1954, 1956, 1960, 1962, 1964, 1970, 1972, 1982, and 1984 because of the stitches entered into the play space 1900 to cause Newton's square root algorithm to operate as the underlying procedure to the gizmo face 1902.

In summary, Figures 19A-19D illustrate the play space 1900 being configured to apply the method proposed by Newton for approximating the square root of any positive real number. This example demonstrates a loop which involves a conditional gizmo so that iteration over a range of numbers is performed until a goal is reached. That goal is tested by the compare number gizmo 1912 (which determines the numerical relationship between two numbers) and, when the goal is reached, the compare numbers gizmo 1912 activates the end event button 1924 of the gizmo face 1902. The gizmo instances 1906, 1910, 1912, and 1916 form a loop that converges on the desired result according to Newton's method when the last two estimates are the same within an acceptable error range.

The copy number gizmo instance 1910 is used to maintain the previous guess so it can be used in the next iteration. The start event button 1980 is part of a fabric which includes both end event buttons 1976 and 1978. When either of these conditions is activated by the compare numbers gizmo instance 1912, the loop is reentered. Completion of the copy operation activates the divide gizmo instance 1906 which in turn activates the average numbers gizmo 1916 which activates the compare numbers gizmo 1912.

If a desired configuration is not possible because the appropriate procedure module is not available, the user can create a new procedure module to meet the requirements of the desired configuration. In one embodiment, the configurations are created through use of the edit utility application 134.

It should be noted that the system according to the present invention does not require a GUI. The system is useful for composing component-based programs that can be used for network programming models or automatic program generation. For example, relationships between object structures can be defined to build an object model without the graphical user interface referred to previously. However, the system becomes more practical and is best understood when described relative to a graphical user interface. It is to be understood that a GUI represents the underlying functionality of a gizmo, such as the gizmo 300.

Further, when a gizmo operates as a basic building block of a gizmo class, it is sometimes referred to as a procedural object. Of course, a procedural object may be created by a user through the edit utility application 134 and comprises at least one component and at least one procedure module. Of note, the procedural object is sometimes referred to as a "primitive" or a "program procedural object" when it operates as a building block for other procedural objects or gizmo classes. For example, in a GUI environment, the edit utility application 134 may present, on the display 112, a play space or work area in which the user places graphic representations of selected components, procedure modules, and procedural objects or gizmos (i.e., the "building blocks"). Through the edit utility application 134, the user defines relationships among the selected components, procedure modules, and procedural objects by creating stitches between values, and between events. Thus, the user creates an intrinsically parallel computer program using only the graphical user interface. Advantageously, the computer program is generated without concern for syntax and other exacting features that are commonly required in computer programming because the building blocks are error free and combinations of the building blocks do not require syntax.

A system and method of programming according to the present invention is suitable for many diverse applications. For example, the world-wide web (WWW) has created many new opportunities for user developed programs, where current browser implementations are built using windowing GUIs. A programming model according to the present invention is particularly suitable for web programming. Virtual reality and other virtual worlds, such as those found in graphical computer games, present other opportunities for application of the gizmo model to a graphical domain. The realization of the gizmo model in a windowing GUI environment graphically represents a gizmo as a dialog box. Events are displayed as button widgets and value boxes or widgets are used to display or enter values.

It is now appreciated that a method and system according to the present invention provides many programming benefits, including integration, concurrency, quality and modularity. Gizmo procedural objects are designed to be program building blocks. There are no restrictions on how simple or how complex a gizmo's behavior may be. Gizmo programming as described herein does not intrinsically limit the programs that are constructed, where the only limitations are the available primitives and the imagination of the user. Gizmos are building blocks that are used to piece together a program which promotes good programming practice. Primitive gizmos may further be implemented to interface with other software. This allows the user to combine the capabilities of many software packages into a custom program. Conventional system interface protocols (like DDE, IPC, OLE, etc.) may be implemented in gizmos to create foreign interfaces.

Each gizmo behaves as an autonomous entity, in communication with other gizmos with which it is combined. The memory for each gizmo is treated as separate, but is in communication with the memory of other gizmos. This symmetry makes it easy to apply the gizmo model to a network application environment. Communicating gizmos could be memory resident on separate machines connected by a network. The user of the gizmo program need not be concerned with how or where the other gizmos in a system are implemented.

While the user need only be concerned with the apparent conceptual model of gizmo programming, the opportunity for optimization still exists. When an optimizing compiler is added to a gizmo development implementation, the compiler has all the important aspects of strongly typed conventional languages and much more information about programmer intent. Compilation takes place on a gizmo abstraction hierarchy. The top gizmo class is the one which represents the root of the tree. The composed gizmos which implement that class are combined and optimized for speed and space efficiency. This also produces a stable encapsulation of the entire hierarchy and thus aids change-management protocols.

The features of gizmo programming naturally promote quality results. Unlike conventional text-based languages there is no description redundancy or syntactic errors. Gizmo programs are constructed in a way to reduce software failures. Further, the

characteristics of visual programming allows for every effective testing and debugging techniques.

Gizmo programs achieve modularity by being divided up into well defined blocks with clear and consistent interfaces. These interfaces help isolate details of the implementation and reveal important program component relationships. Initially the benefit of modularity makes it easier to modify and maintain a program. Ultimately, modules may be reused by other programmers and programs. Reuse provides for improved programmer productivity and product quality. Gizmo programming according to the present invention encourages good modularity. Each gizmo represents a module that can be combined with other modules to define a program. The added benefit is that a gizmo can be initiated without any context. Each gizmo is instantiated in a way that makes it function as a stand alone program. Even if incorrect values are provided, the gizmo responds with the activation of an error end event.

Another important gizmo reuse feature is the lack of dependence on names. In windowing or GUI forms of gizmo implementation, naming is merely a decoration. The user refers to the elements of a program by pointing and relates them by stitching. Name compatibility is no longer a reuse consideration. The gizmo paradigm provides a mechanism for truly modular reusable functionality. It reveals the true utility of the computer directly to users, and results in unfettered creativity and cooperation among users.

The above-listed sections and included information are not exhaustive and are only exemplary for certain computer systems. The particular sections and included information in a particular embodiment may depend upon the particular implementation and the included devices and resources. Although a system and method according to the present invention has been described in connection with the preferred embodiment, it is not intended to be limited to the specific form set forth herein, but, on the contrary, it is intended to cover such alternatives, modifications, and equivalents as can be reasonably included within the spirit and scope of the invention as defined by the appended claims.

CLAIMS

1. A computer programming system, comprising:

a plurality of components, each component associated with input/output and comprising either one of a value and an event;

a plurality of procedures, each procedure associated with at least one component including at least one input event;

at least one combiner, where each combiner is used to link any values together to form stitched values and to link any events together to form stitched events; and

an executor that detects activation of a first input event, that executes any of the plurality of procedures upon activation of the at least one associated input event, that updates other stitched values if any one value of the any stitched values is updated and that activates stitched events if any one event of the any stitched events is activated.

2. The computer programming system of claim 1, further comprising:

each of the plurality of procedures, if and when executed by the executor, receiving any associated input values, detecting any associated input events, and, as determined by the associated procedure, updating any associated output values and activating any associated output events.

3. The computer programming system of claim 1, further comprising:

a plurality of procedural objects, each comprising at least one procedure and at least one component associated with the at least one procedure.

4. The computer programming system of claim 3, further comprising:

the at least one combiner linking at least one component of a first procedural object to at least one component of a second procedural object.

5. The computer programming system of claim 4, further comprising:

the at least one combiner linking at least one output event of a first procedural object to at least one input event of a second procedural object and to at least one input event of a third procedural object; and

the second procedural object including a first procedure and the third procedural object including a second procedure.

6. The computer programming system of claim 5, wherein the first procedure and second procedure are effectively executed simultaneously by the executor.

7. The computer programming system of claim 3, further comprising:

at least one of the plurality of procedural objects including at least one input component and at least one output component.

8. The computer programming system of claim 7, further comprising:

the at least one of the plurality of procedural objects including at least one output value having a value that depends upon the at least one input component when the at least one procedure of the at least one of the plurality of procedural objects is executed.

9. The computer programming system of claim 7, further comprising:

the at least one of the plurality of procedural objects including at least one output event that is activated depending upon the at least one input component when the at least one procedure of the at least one of the plurality of procedural objects is executed.

10. The computer programming system of claim 3, further comprising:

each of the plurality of procedural objects including at least one component linked with at least one component of another of the plurality of procedural objects.

11. The computer programming system of claim 10, further comprising:

each of the plurality of procedural objects being self-timed and coherent.

12. The computer programming system of claim 10, further comprising:

each of the plurality of procedural objects, when executed, operating independently with respect to every other of the plurality of procedural objects.

13. The computer programming system of claim 1, wherein the executor operates as an interpreter.

14. The computer programming system of claim 1, further comprising:

the executor including a compiler that generates executable program code based on the plurality of procedures, the plurality of components and any linking of components.

15. A computer system, comprising:

a processor;

at least one input device coupled to the processor;

a memory, coupled to the processor and the at least one input device, that stores data and program code for execution by the processor, the program code including:

a plurality of component objects, each component object comprising either one of a value and an event;

a plurality of procedure modules, each procedure module associated with at least one component object;

an edit utility that when executed by the processor, enables a user via the at least one input device to stitch component objects of selected procedure module instances together to form a new procedural object and to store the new procedural object in the memory; and

the processor, when executing the new procedural object, executing any of the selected procedure modules included in the new procedural object, updating any stitched values if any one value of the any stitched values is updated and activating any stitched events if any one event of the any stitched events is activated.

16. The computer system of claim 15, further comprising:

a display coupled to the processor, the at least one input device and the memory;

the edit utility displaying graphic representations of selected procedure modules and component objects on the display; and

the edit utility displaying graphic representations of manipulations of the at least one input device by the user on the display to enable the user to interactively stitch component objects of the selected procedure modules together to create the new procedural object.

17. The computer system of claim 16, further comprising:

the edit utility enabling a user to associate a procedure graphic with the new procedural object; and

the executor displaying the procedure graphic and enabling the user to execute the new procedural object by interacting with the procedure graphic via the at least one input device.

18. The computer system of claim 15, further comprising:

the memory storing a plurality of predetermined procedural objects, each comprising at least one procedure module and at least one component object associated with the at least one procedure module.

19. The computer system of claim 18, further comprising:

the plurality of predetermined procedural objects comprising a library of predefined procedural object primitives, each that performs at least one basic programming function.

20. The computer system of claim 18, further comprising:

the edit utility enabling a user to associate a procedure block with the new procedural object; and

the executor enabling the user to execute the new procedural object by accessing the procedure block via the at least one input device.

21. The computer system of claim 20, further comprising:

the memory storing the new procedural object;

the edit utility enabling the user, via the at least one input device, to retrieve instances of any of the predetermined procedural objects and to retrieve an instance of the new procedural object as represented by the procedure block; and

the edit utility enabling the user to stitch component objects of the new procedural object with component objects of retrieved instances of any of the predetermined procedural objects to create a second new procedural object and to store the second new procedural object in the memory.

22. The computer system of claim 18, further comprising:

a display coupled to the processor, the at least one input device and the memory;

the edit utility displaying, on the display, graphic representations of retrieved procedural object instances including graphic representations of any component objects associated with each retrieved procedural object instance; and

the edit utility displaying representations of inputs and manipulations of the at least one input device by the user on the display to enable the user to interactively stitch component objects of the each retrieved procedural object together to create the new procedural object.

23. The computer system of claim 22, further comprising:

the edit utility displaying a procedure graphic and enabling a user to associate the program graphic with the new procedural object; and

the executor displaying the procedure graphic and enabling the user to execute the new procedural object by interacting with program graphic via the at least one input device.

24. The computer system of claim 23, further comprising:

the memory storing the new procedural object;

the edit utility enabling the user, via the at least one input device, to retrieve and display graphic representations of any of the plurality of predetermined procedural objects and an instance of the new procedural object as represented by the procedure graphic; and

the edit utility enabling the user, via the at least one input device, to stitch component objects of the new procedural object represented by the procedure graphic with component objects of the any of the plurality of predetermined procedural objects retrieved by the user as represented by the graphic representations to create a second new procedural object and to store the second new procedural object in the memory.

25. The computer system of claim 15, further comprising:

an interpreter that interprets the plurality of procedure modules and stitched components of the new procedural object when executed; and

the processor executing the new procedural object via the interpreter.

26. The computer system of claim 15, further comprising:

a compiler that compiles the new procedural object into executable program code; and the processor accessing and executing the program code.

27. A distributed computation system, comprising:

a network;

a plurality of computers participating in the network, each computer comprising a corresponding one of a plurality of processors and a corresponding one of a plurality of memory systems;

the plurality of processors and the plurality of memory systems comprising a distributed processing system;

the plurality of computers including a first computer, the first computer comprising:

a first processor;

at least one first input device coupled to the first processor;

a first memory system, coupled to the first processor and the at least one first input device, that stores program code for execution by the first processor, the program code including:

a plurality of component objects, each component object comprising either one of a value and an event;

a plurality of procedure modules, each procedure module associated with at least one component object; and

an edit utility that when executed by the first processor, enables a user via the at least one first input device to retrieve any of the plurality of procedure modules and associated component objects, to stitch component objects of retrieved procedure modules together to create a new procedural object and to store the new procedural object in the first memory system; and

the first processor that executes the new procedural object by distributing the plurality of procedure modules included in the new procedural object among the distributed processing system; and

each processor of the distributed processing system executing received procedure modules, updating any received stitched values if any one value of the any stitched values is

updated and activating any received stitched events if any one event of the any stitched events is activated.

28. A method of programming a computer system, comprising:

retrieving instances of selected procedural objects from a plurality of procedural objects, each of the plurality of procedural objects including an associated procedure and at least one associated input/output component, each of the at least one associated input/output component comprising one of an event and a value and the at least one associated input/output component including at least one associated input event that initiates an associated procedure;

stitching together components of the instances of the selected procedural objects including at least one start event that invokes an associated procedure of at least one of the selected procedural objects when the at least one start event is activated; and

storing a new procedural object that includes the instances of the selected procedural objects and fabric representing components that are stitched together.

29. The method of claim 28, wherein the stitching comprises including a combiner instance for each stitch that links components of the instances.

30. The method of claim 28, further comprising:

the retrieving including displaying a graphic of each of the instances of selected procedural objects; and

for each of the instances of selected procedural objects, including a graphic for each of the at least one associated component.

31. The method of claim 30, wherein the stitching includes displaying stitch graphics to facilitate a visual representation of stitching components and of stitched components.

32. The method of claim 28, further comprising:

a processing means executing the new procedural object by interpreting the new procedural object to detect any of the at least one start event, to execute any associated procedure of the instances of selected procedural objects initiated by the associated at least one input event, to update other stitched values if any one value of the any stitched values is

updated, and to activate stitched events if any one event of the any stitched events is activated.

33. The method of claim 28, further comprising:

compiling the new procedural object into an executable program, the executable program including code to detect any of the at least one start event, code of any associated procedure of the instances of selected procedural objects, code to update other stitched values if any one value of the any stitched values is updated, and code to activate stitched events if any one event of the any stitched events is activated.

34. The method of claim 28, further comprising:

retrieving the new procedural object; and

in an edit mode, modifying the fabric by modifying the stitches between any components.

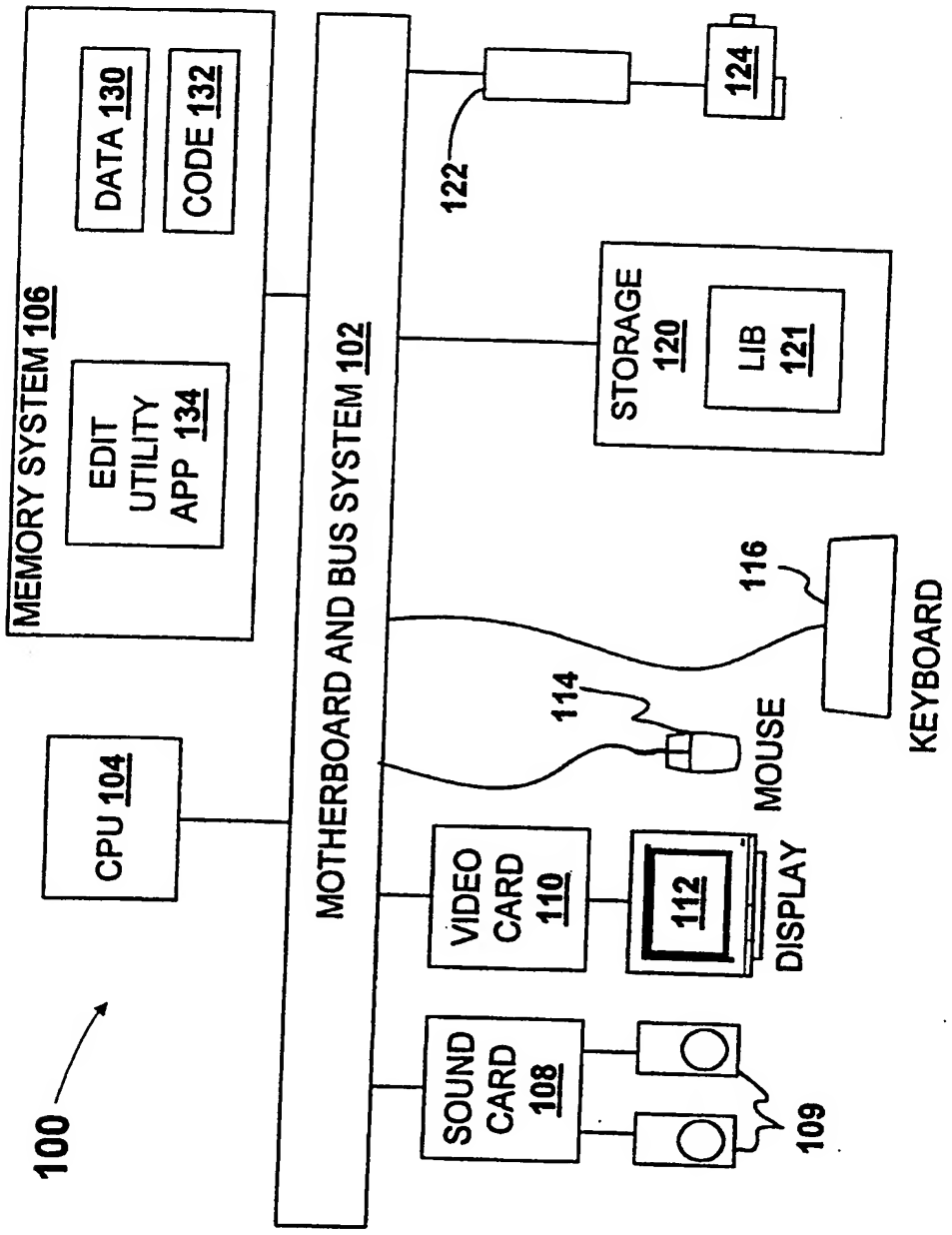

100
MEMORY SYSTEM 106
DATA 130
CODE 132
EDIT UTILITY APP 134
CPU 104
MOTHERBOARD AND BUS SYSTEM 102
122
124
STORAGE 120
LIB 121
116
KEYBOARD
114
MOUSE
VIDEO CARD 110
112
DISPLAY
SOUND CARD 108
109

FIG. 1

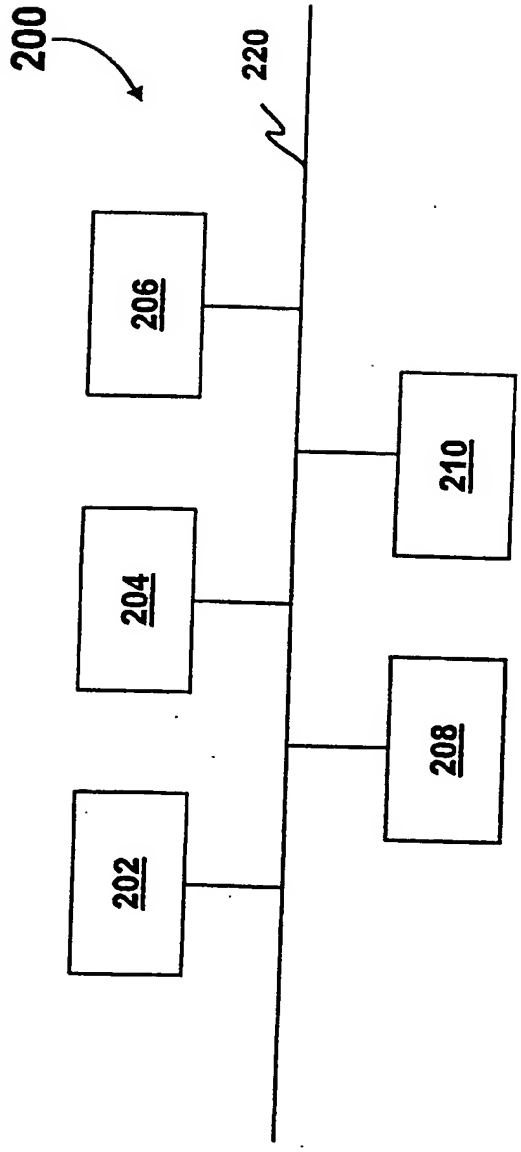


FIG. 2

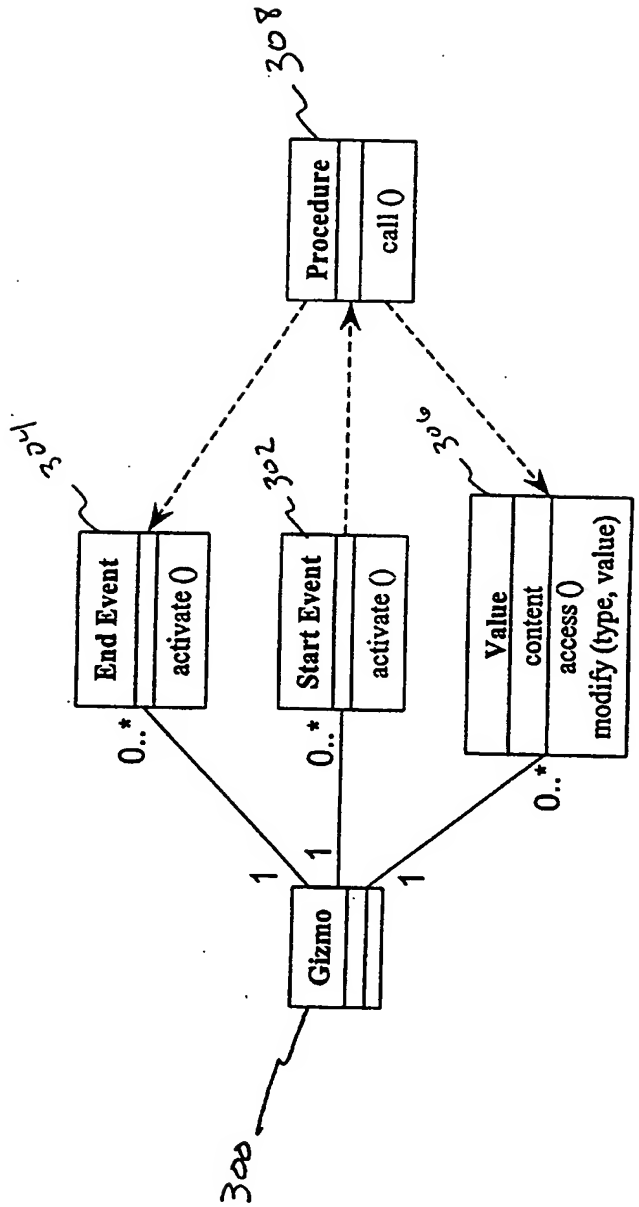

308
Procedure
call ()
304
End Event
activate ()
302
Start Event
activate ()
306
Value
content
access ()
modify (type, value)
0..*
0..*
0..*
1
1
1
Gizmo
300

Figure 3

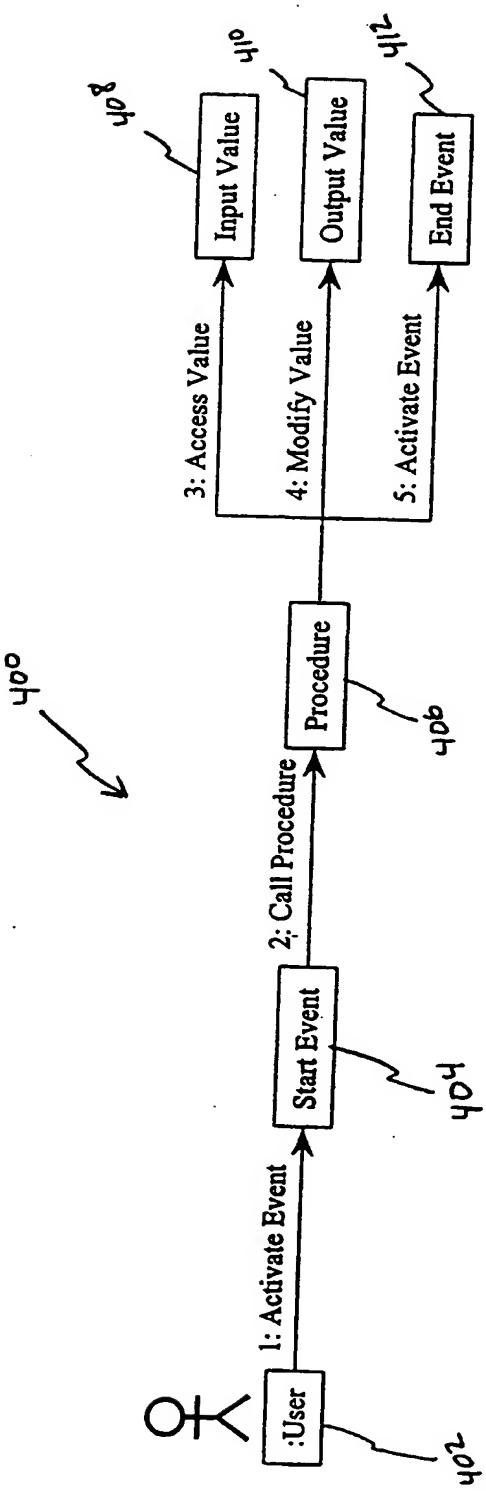

400
:User
402
1: Activate Event
Start Event
404
2: Call Procedure
Procedure
406
3: Access Value
4: Modify Value
5: Activate Event
Input Value
408
Output Value
410
End Event
412

Figure 4

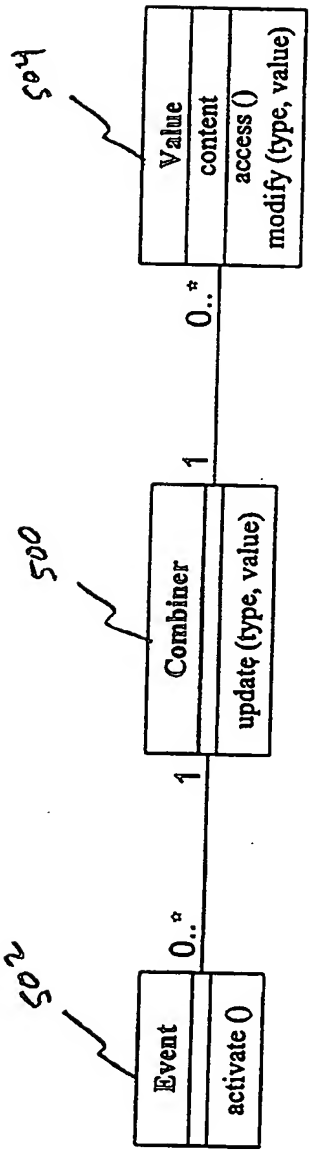

504
Value
content
access ()
modify (type, value)
0..*
1
500
Combiner
update (type, value)
1
0..*
502
Event
activate ()

Figure 5

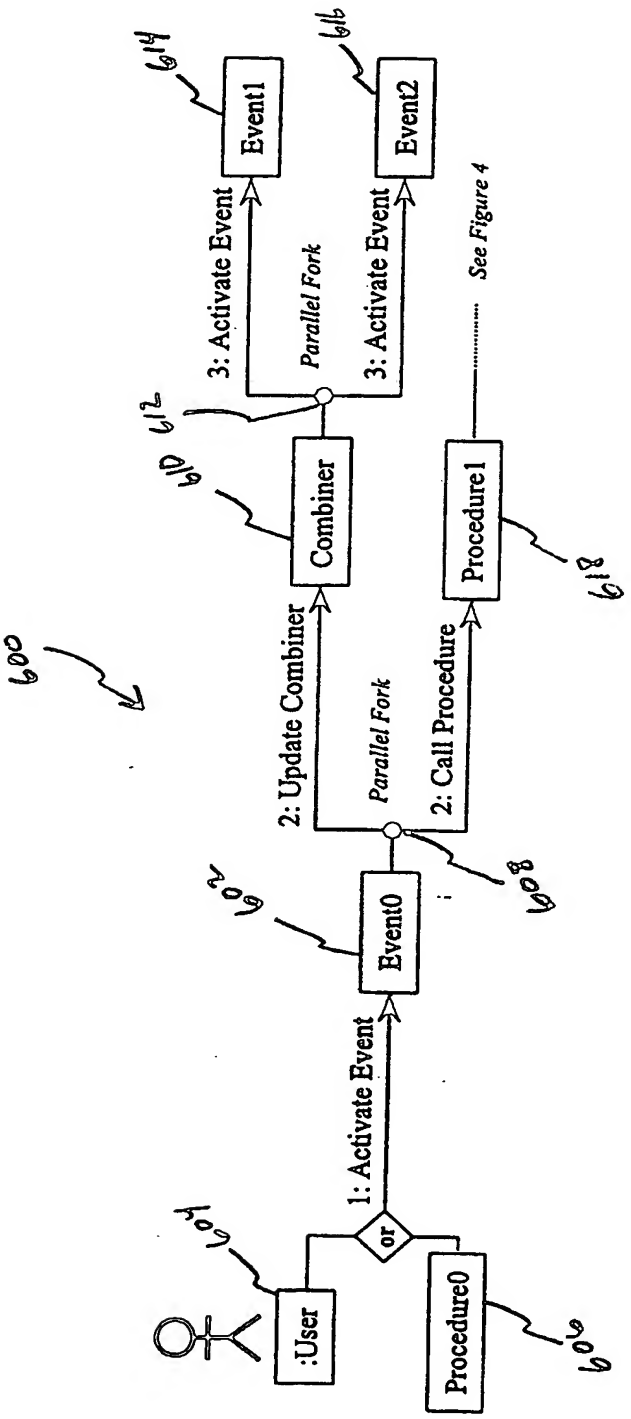

600
604
:User
or
1: Activate Event
606
Event0
2: Update Combiner
2: Call Procedure
Parallel Fork
608
Combiner
Procedure1
See Figure 4
610
618
612
3: Activate Event
3: Activate Event
Parallel Fork
614
Event1
616
Event2
Procedure0

Figure 6

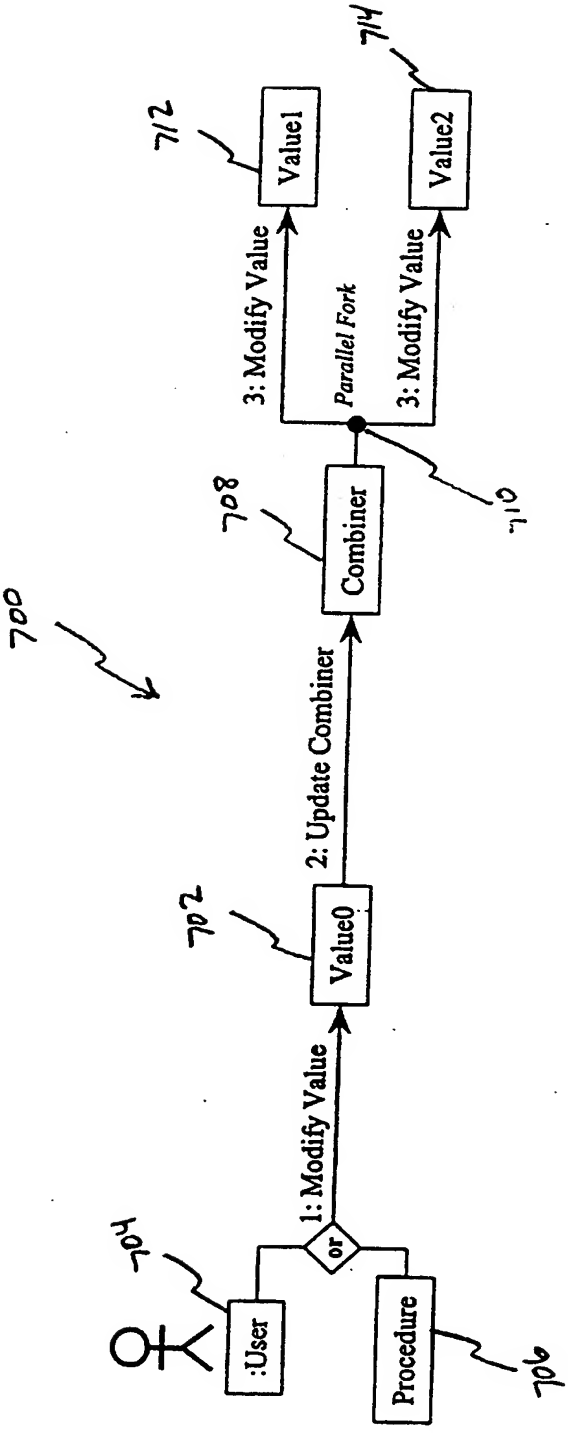

700
:User
704
Procedure
706
or
1: Modify Value
Value0
702
2: Update Combiner
Combiner
708
Parallel Fork
710
3: Modify Value
3: Modify Value
Value1
712
Value2
714

Figure 7

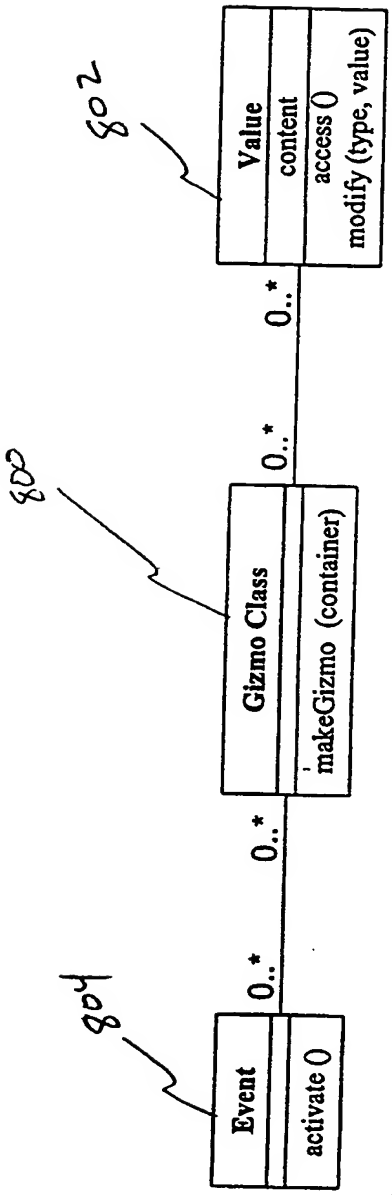

802
Value
content
access ()
modify (type, value)
0..*
0..*
800
Gizmo Class
makeGizmo (container)
0..*
0..*
804
Event
activate ()

Figure 8

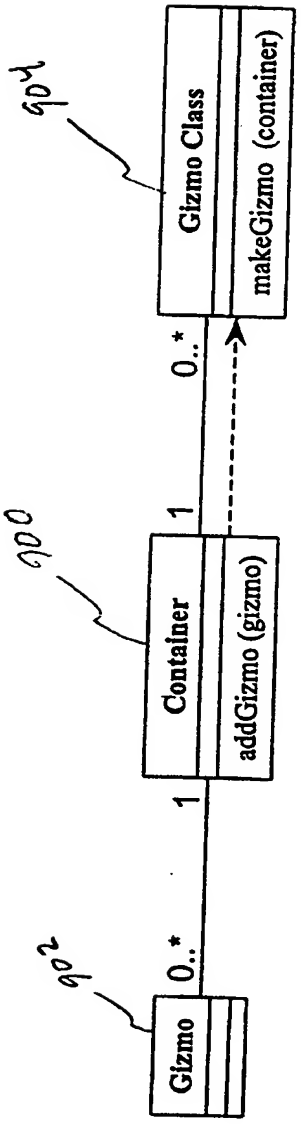

Gizmo Class
makeGizmo (container)
Container
addGizmo (gizmo)
Gizmo
0..*
1
1
0..*
104
100
102

Figure 9

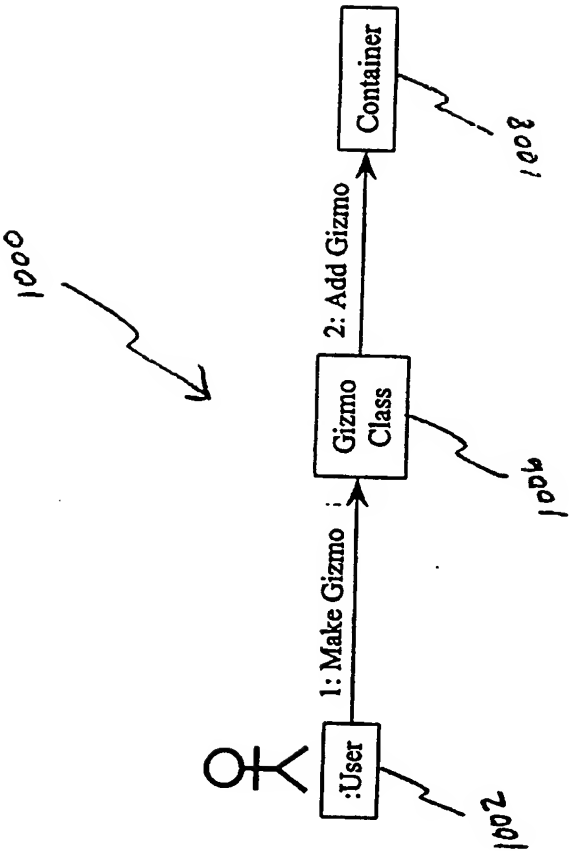

1000
Container
1008
2: Add Gizmo
Gizmo Class
1006
1: Make Gizmo
:User
1002

Figure 10

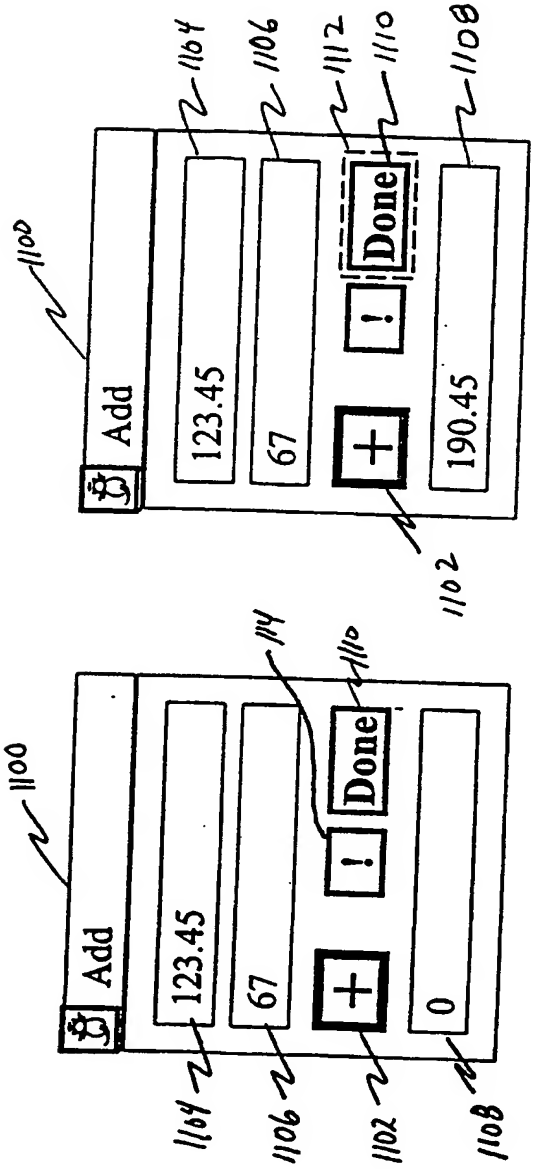


Figure 11A

Figure 11B

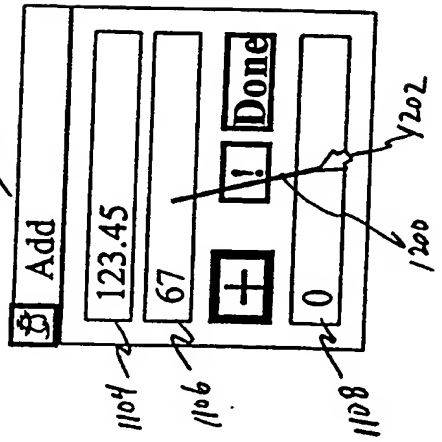


Figure 12A

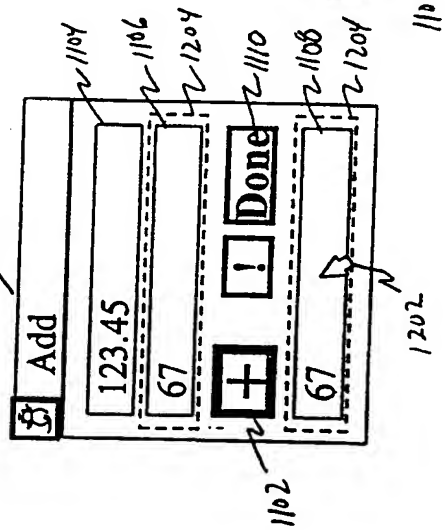


Figure 12B

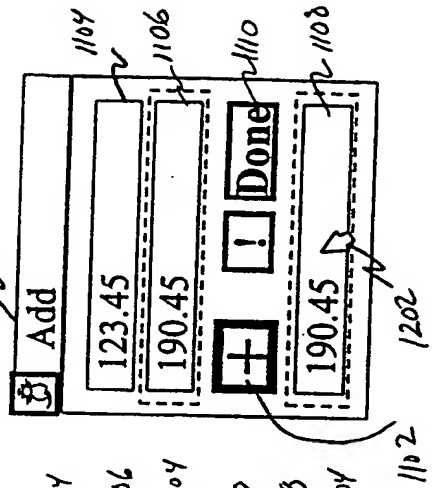


Figure 12C

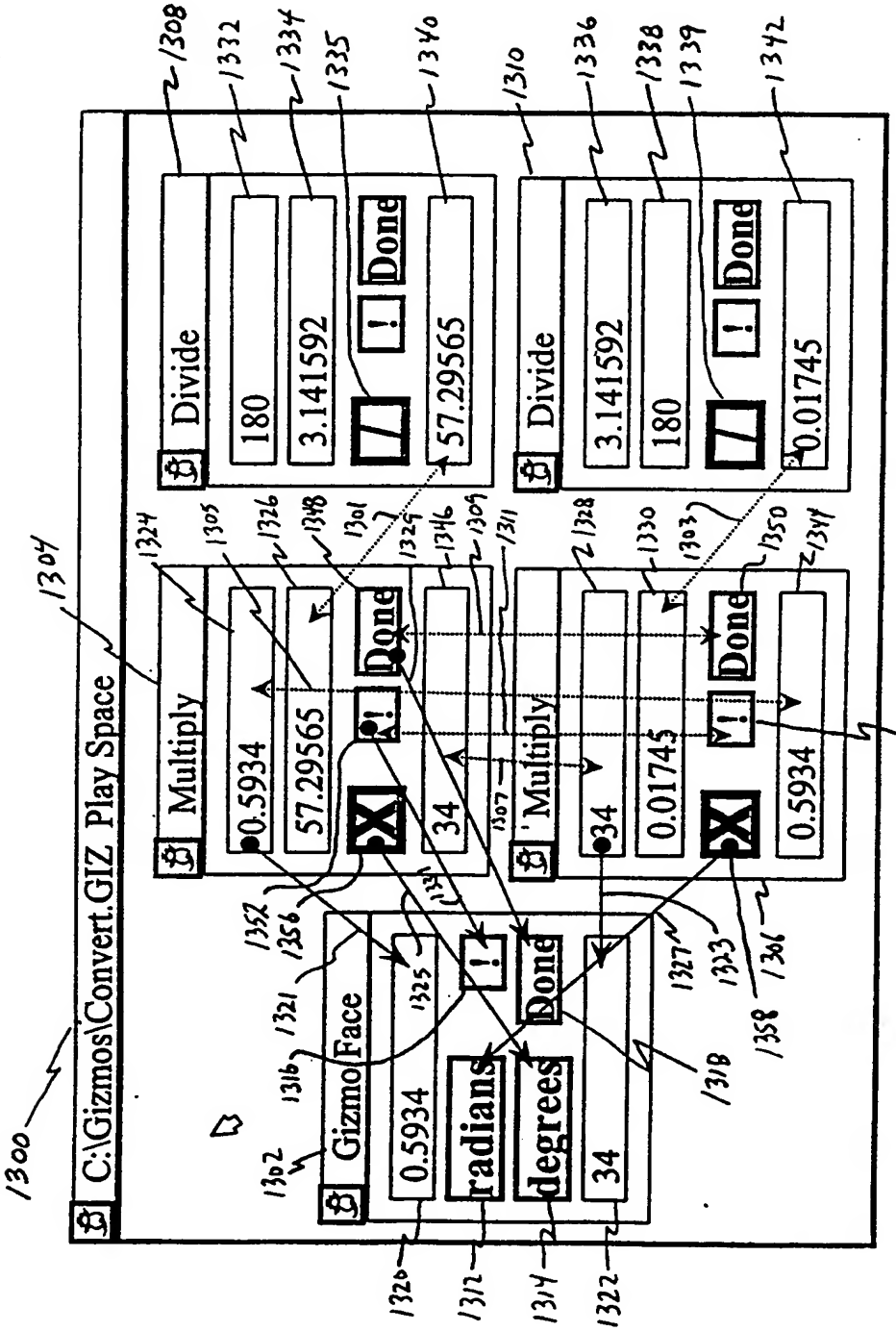

C:\Gizmos\Convert.GIZ Play Space
Gizmo Face
0.5934
radians
degrees
34
Multiply
0.5934
57.29565
34
Done
Multiply
34
0.01745
0.5934
Done
Divide
180
3.141592
Done
57.29565
Divide
3.141592
180
Done
0.01745

Figure 13

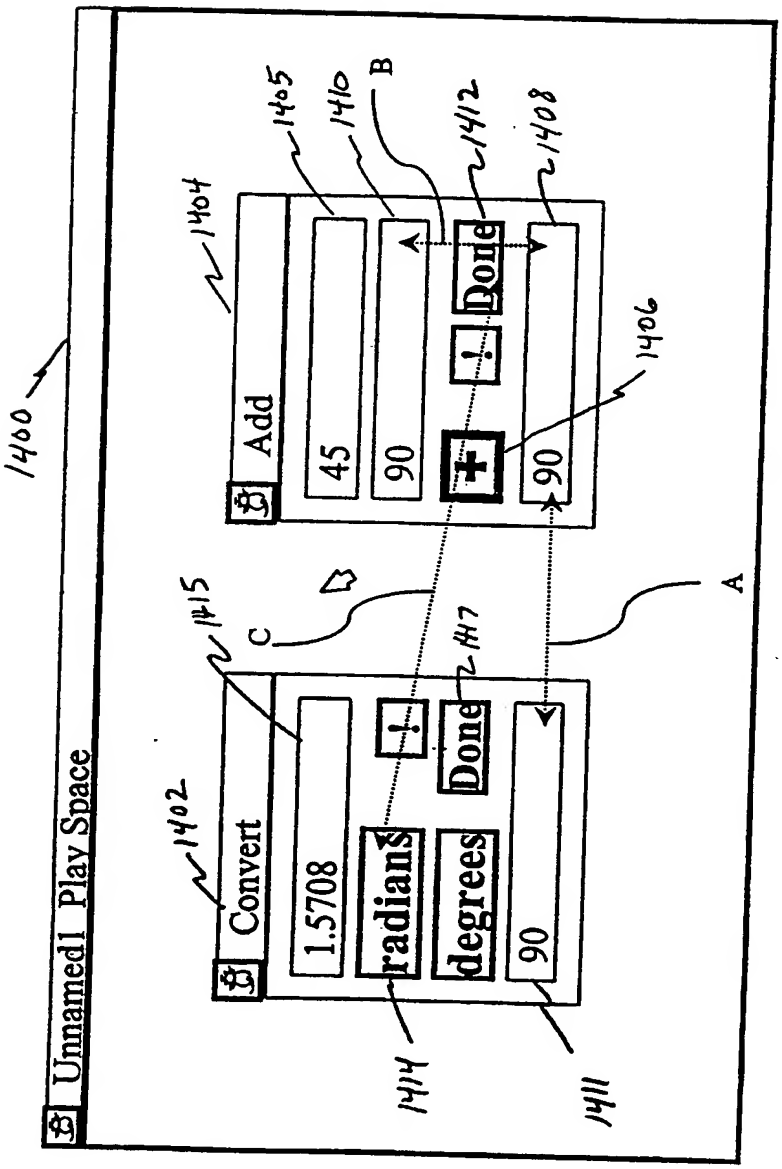

Unnamed1 Play Space
1400
Add
45
90
+
!
Done
90
1404
1405
1410
B
1412
1408
1406
Convert
1.5708
radians
degrees
90
!
Done
1402
1415
C
1417
A
1414
1411

Figure 14

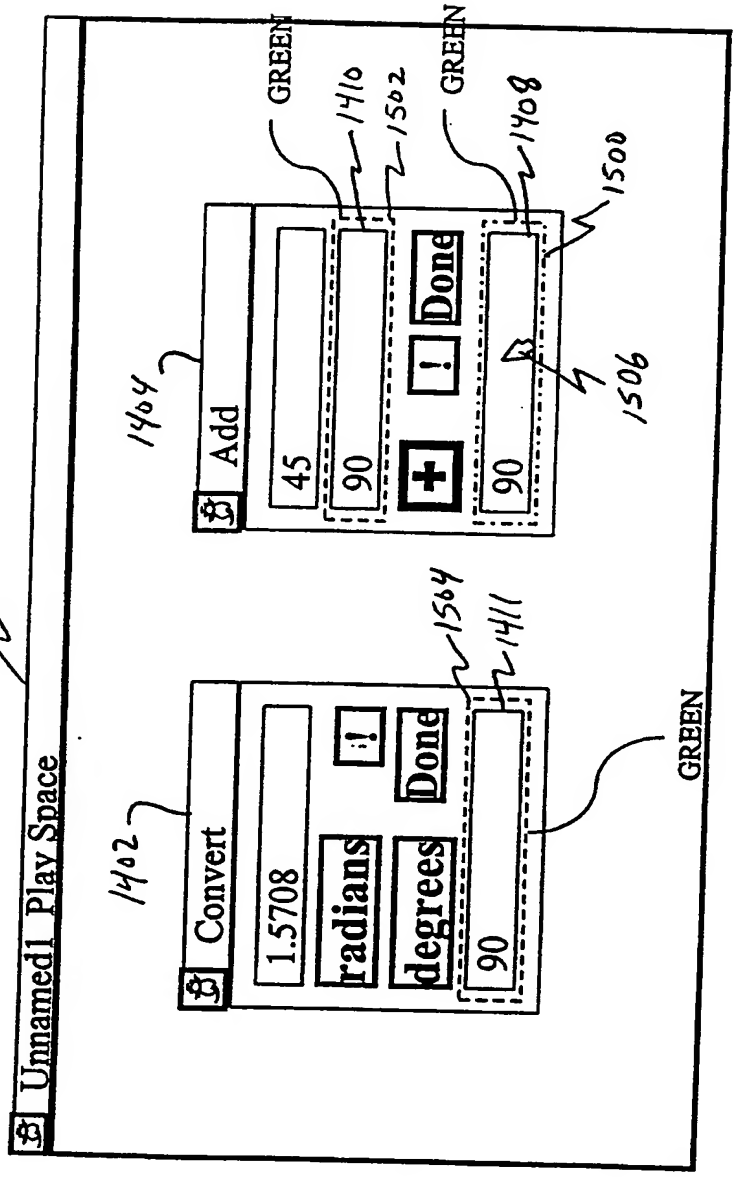

Unnamed1 Play Space
1400
Add
1404
45
90
GREEN
1410
1502
Done
GREEN
1408
1500
90
1506
Convert
1402
1.5708
radians
degrees
Done
1504
1411
90
GREEN

Figure 15

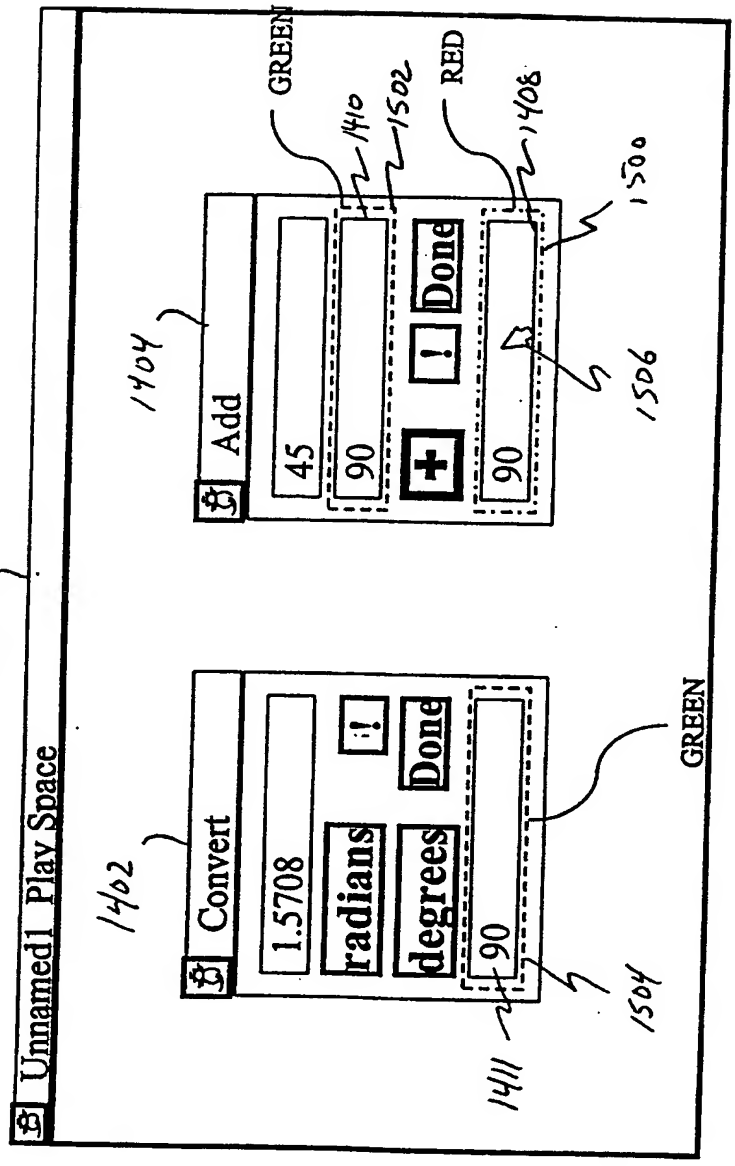

Unnamed1 Play Space
1400
Convert
1.5708
radians
degrees
Done
90
1402
1411
1504
GREEN
Add
45
90
1404
1410
1502
GREEN
RED
1408
1500
1506

Figure 16

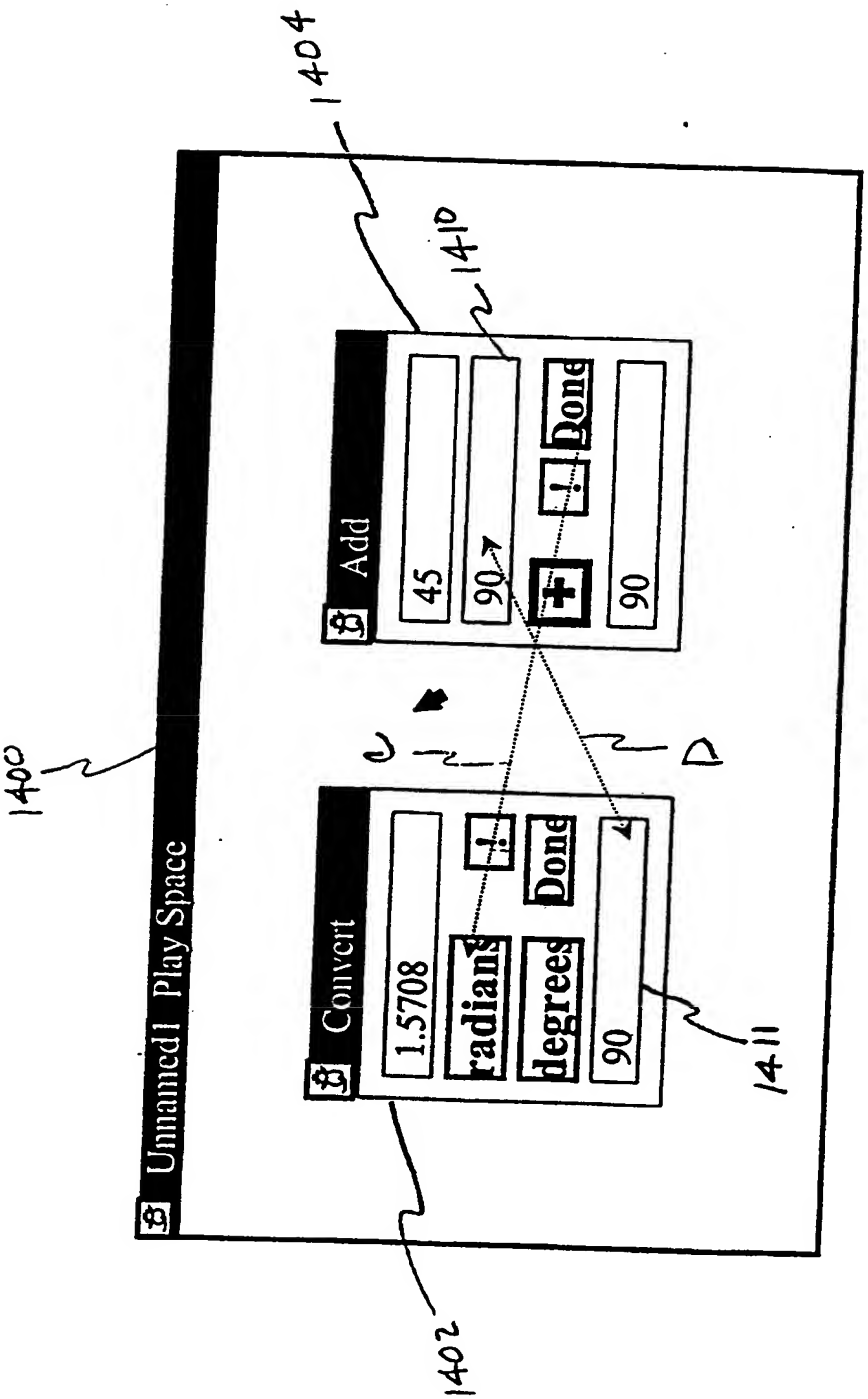

1404
1410
1400
Unnamed1 Play Space
Add
45
90
Done
90
Convert
1.5708
radians
degrees
Done
90
1411
1402

Figure 17

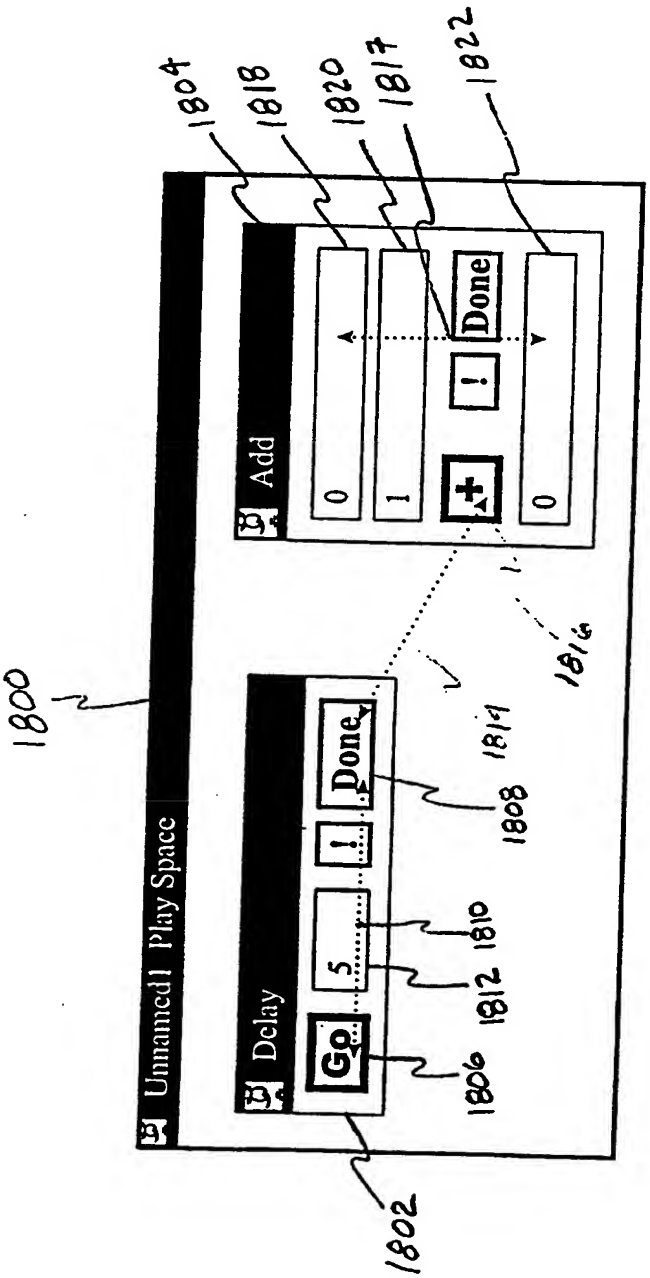

1800
1802
1804
1806
1808
1810
1812
1814
1816
1817
1818
1820
1822
Unnamed1 Play Space
Delay
Go
5
!
Done
Add
0
1
+
!
Done
0

Figure 18

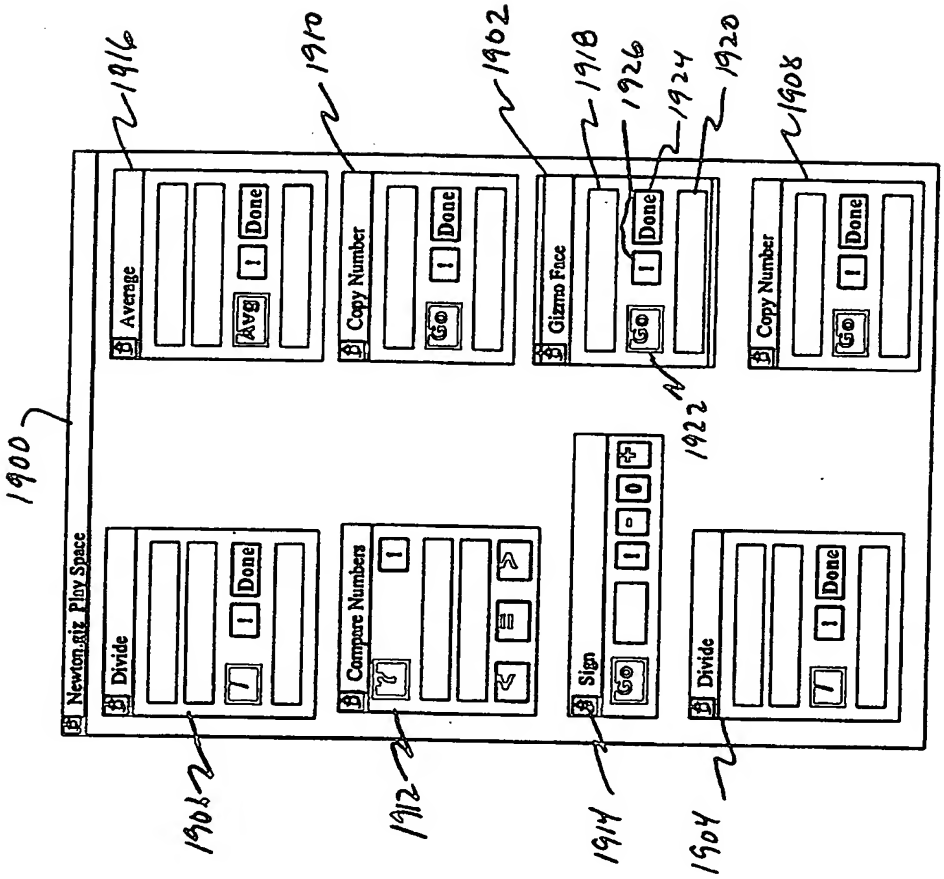

1900
Newton.stz Play Space
Divide
Done
Compare Numbers
Sign
Divide
Done
Average
Done
Avg
Copy Number
Done
Go
Gizmo Face
Done
Go
Copy Number
Done
Go
1916
1910
1902
1918
1926
1924
1920
1908
1922
1906
1912
1914
1904

Figure 19A

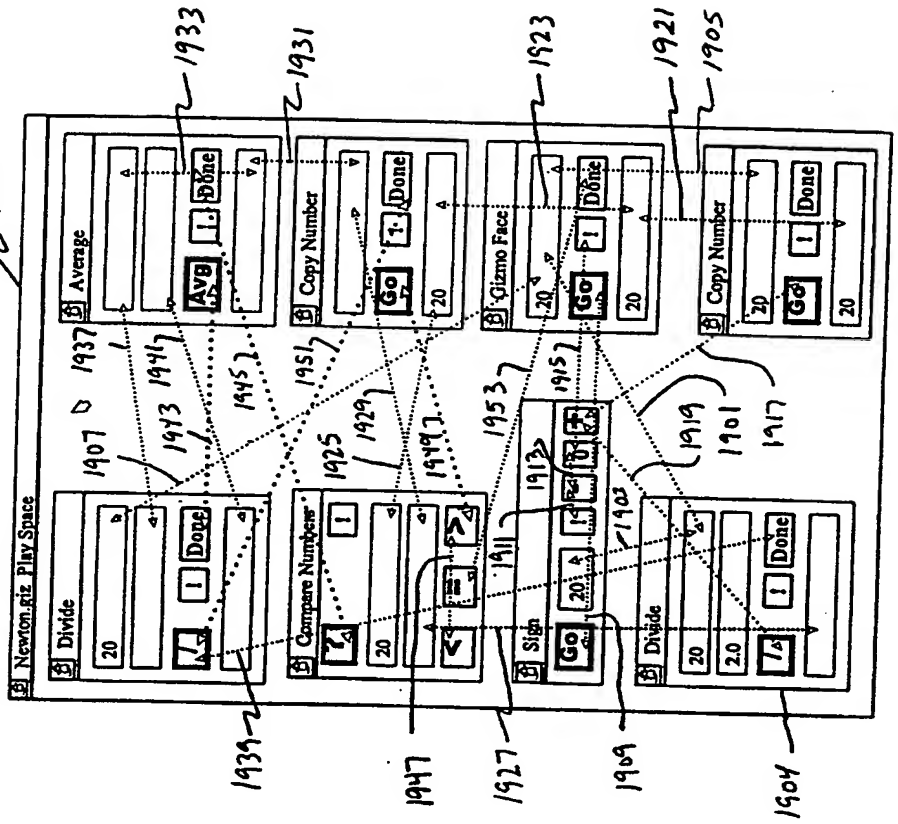

1900
Newton.nit Play Space
Average
Done
Ave
Copy Number
Go
Gizmo Face
Divide
Compare Numbers
Sign
1933
1931
1923
1921
1905
1937
1941
1945
1951
1907
1973
1925
1929
1949
1953
1915
1919
1901
1917
1911
1903
1939
1947
1927
1909
1904

Figure 19B

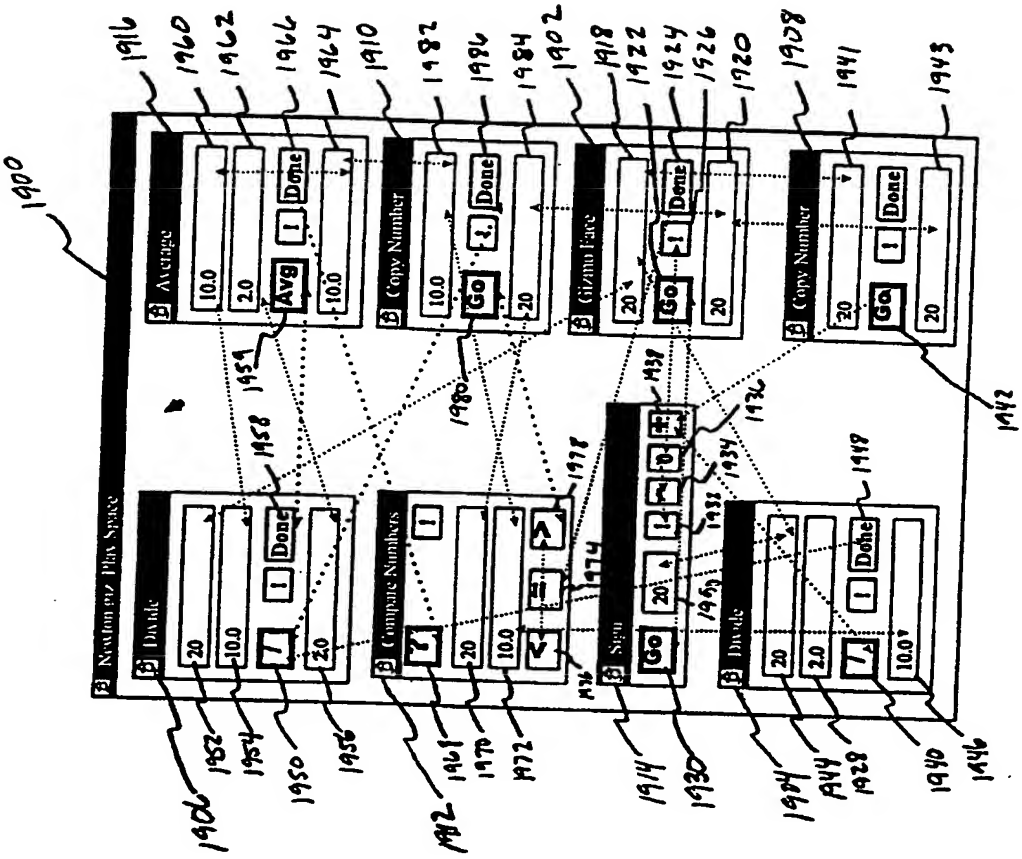

Newtonian Play Space
Divide
Average
Compute Numbers
Sign
Gizmo Face
Copy Number
Copy Number
Divide
Done
Avg
Go
1900
1916
1960
1962
1966
1964
1910
1982
1986
1984
1902
1918
1922
1924
1926
1920
1908
1941
1943
1958
1959
1980
1978
1974
1938
1936
1934
1932
1919
1912
1904
1952
1954
1950
1956
1982
1965
1970
1972
1914
1930
1944
1944
1928
1940
1946

Figure 19C

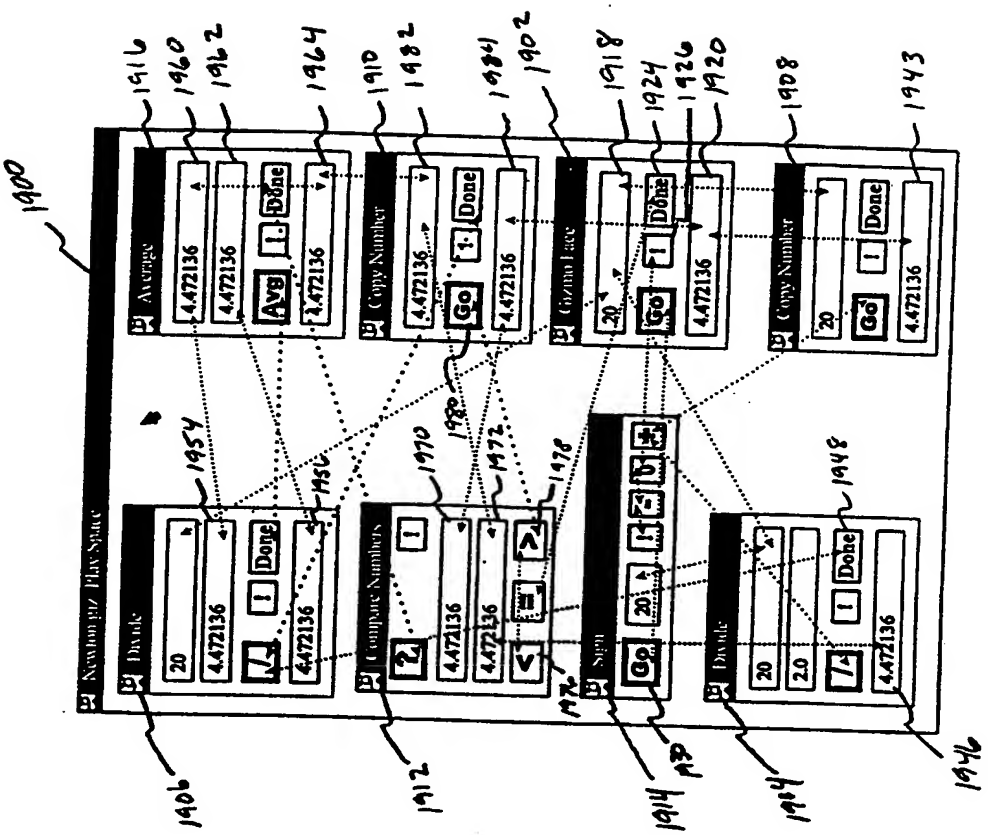

1900
1916
1960
1962
1964
1910
1982
1984
1902
1918
1924
1926
1920
1908
1943
Average
Copy Number
Ozone Layer
Copy Number
Done
Go
Avg
1954
1956
1970
1972
1978
1988
Newton gz Play Space
Divide
Compare Numbers
Sqrt
Divide
1948
1906
1912
1914
1914
1946

Figure 19D